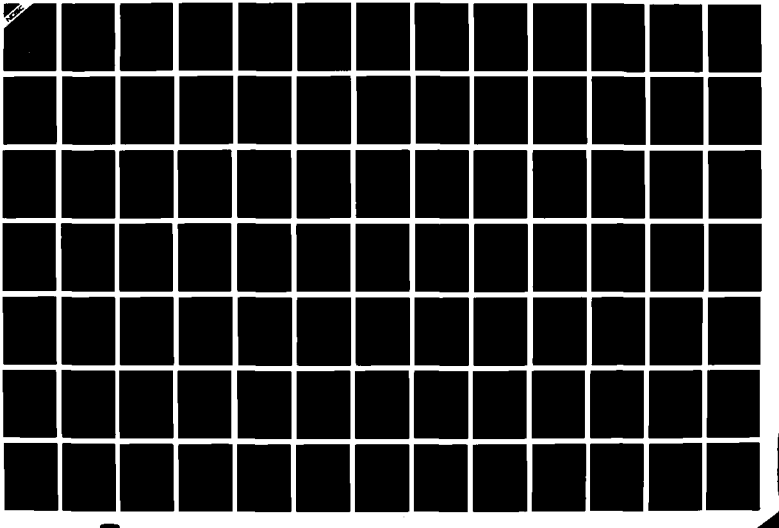
AD-A184 491

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
CONCEPTS & TECHNIQUES FOR SUPPORT OF REAL TIME DISTRIBUTED OPERATING SYSTEMS
BY: CARNEGIE-MELLON UNIVERSITY

1 OF 3
NOSC TD 1113
UNCLASSIFIED
JUL 87



NOSC TD 1113

NOSC TD 1113

NAVAL OCEAN SYSTEMS CENTER San Diego, California 92152-5000

Technical Document 1113
July 1987

Concepts and Techniques for Support of Real Time Distributed Operating Systems

Computer Science Department
Carnegie-Mellon University

Approved for public release;
distribution is unlimited.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Naval Ocean Systems Center or the U.S. government.

NAVAL OCEAN SYSTEMS CENTER

San Diego, California 92152-5000

E. G. SCHWEIZER, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This report was prepared by Carnegie - Mellon University under contract N66001-83-C-0305 for Code 443 of the Naval Ocean Systems Center, San Diego, CA 92152-5000.

Released by
D.C. McCall, Head
C^2 Information Processing
and Display Technology
Branch

Under authority of
W.T. Rasmussen, Head
Advanced C^2 Technologies
Division

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S) NOSC TD 1113
6a NAME OF PERFORMING ORGANIZATION Carnegie - Mellon University 6b OFFICE SYMBOL (if applicable)	7a NAME OF MONITORING ORGANIZATION Naval Ocean Systems Center
6c ADDRESS (City, State and ZIP Code) Computer Sciences Department Pittsburgh, PA 15213	7b ADDRESS (City, State and ZIP Code) San Diego, CA 92152-5000
8a NAME OF FUNDING SPONSORING ORGANIZATION Office of Naval Technology 8b OFFICE SYMBOL (if applicable) ONT	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N66001-83-C-0305
8c ADDRESS (City, State and ZIP Code) 800 N. Quincy Arlington, VA 22217-5000	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 61153N; PROJECT NO: CC15; TASK NO:; AGENCY ACCESSION NO: DN488 752
11 TITLE (Include Security Classification) Concepts and Techniques for Support of Real Time Distributed Operating Systems	
12 PERSONAL AUTHOR(S)	
13a TYPE OF REPORT Interim 13b TIME COVERED FROM Feb 85 TO Dec 85	14 DATE OF REPORT (Year, Month, Day) July 1987 15 PAGE COUNT 264
16 SUPPLEMENTARY NOTATION	
17 COSATI CODES FIELD / GROUP / SUB-GROUP	18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) compound transaction freeze mode unfreeze process

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The goal of the Archons project is to conduct research into the issues of decentralization in distributed computing systems at the operating level and below. The primary research issues being investigated within Archons are decentralized control team and consensus decision making, transaction management, probabilities algorithms, and architectural support in a real-time environment.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL D. Small	22b TELEPHONE (Include Area Code) (619)225-7196 22c OFFICE SYMBOL Code 443

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

List of Figures

SECTION I

1. Introduction

This document records interim progress during the period 1 February 1985 through31 December 1985 since the last technical report. Contained in this report are three major design specifications for the ArchOS decentralized operating system that is being built at CMU as part of the Archons project with partial support from NOSC. Each design specification appears as a separate chapter in Section II of this report.

The Client Interface Specification in Chapter 4 of this report describes the functionality of ArchOS as seen by client application programs. In Chapter 5, the ArchOS System Architecture Specification summarizes the structure of ArchOS that supports the ArchOS client interface. Chapter 6 contains the System Design Description of ArchOS. The System Design describes the structure and functionality of major components of ArchOS, such as the ArchOS kernel, the arobject/process management subsystem, the communication subsystem, the transaction subsystem, the file subsystem, the I/O device subsystem, the time-driven scheduler subsystem, and the monitoring and debugging subsystems.

This document does not report in detail on the current status and design of the Alpha kernel software that will eventually support ArchOS in the Archons Testbed. A separate report devoted to Alpha is currently being prepared and will be delivered shortly.

2. Summary

The goal of the Archons project is to conduct research into the issues of decentralization in distributed computing systems at the operating system level and below. The primary research issues being investigated within Archons are decentralized control, team and consensus decision making, transaction management, probabilistic algorithms, and architectural support in a real-time environment.

A major focus of research in Archons is the ArchOS decentralized operating system. ArchOS will serve as the primary evidence that the theoretical ideas and practical implementation approaches proposed by Archons actually perform satisfactorily. ArchOS consists of the ArchOS kernel, subsystems, and system objects. The ArchOS kernel provides basic mechanisms for time-driven resource management. The subsystems together with system objects establish ArchOS facilities.

The primary results achieved by CMU's effort during 1985 are as follows:

- completion of the Client Interface Specification for ArchOS,
- completion of the System Architecture Specification for ArchOS,
- completion of the System Design Description for ArchOS, and
- successful implementation and testing of the initial version of the Alpha kernel.

3. Conclusions

Continued and steady progress is being achieved in developing the ArchOS decentralized operating system. The design documents produced to date, in conjunction with the kernel software actually constructed, together represent a substantial step towards realization of a real-time decentralized operating system based on objects and transactions. Further progress depends on a combination of experimentation with concepts proposed in the design documentation and integration of kernel software with higher level operating system software and resource management strategies. The Archons project is incrementally achieving the objective of developing a decentralized operating system that embodies novel concepts suitable for application in Navy real-time command and control systems.

SECTION II

4. Client Interface Specification

4.1 Overview

ArchOS is the operating system being designed and constructed as part of the Archons research project. The stated goals of the project are to conduct research into the issues of decentralization in distributed computing systems at the operating system level and below. In this area, the primary research issues being investigated within Archons are decentralized control, team decision making, transaction management, probabilistic algorithms, and architectural support.

It is planned that ArchOS will be developed in three stages. The first, called the interim testbed version, is now underway and will handle a small set of Sun workstations[1] interconnected by an Ethernet. It is expected that this operating system will constitute an existence proof of many of the basic concepts involved in our research as applied to the construction of a decentralized operating system. This system will later be followed by a more complete testbed operating system incorporating the lessons learned in the interim testbed construction. Finally, an analysis of the ArchOS structure is expected to result in the construction of specialized hardware for the purpose of executing a full-blown ArchOS system, and ArchOS will be rewritten to run on it at that time.

In this document, we describe the characteristics of the interim testbed version of ArchOS currently being designed. This system is intended to be a vehicle with which these issues can be investigated, rather than an operational applications programming environment. The client interface, then, must be have sufficient functional completeness to allow test applications to be constructed with which to study ArchOS' characteristics, but need not provide a particularly complete set of facilities. The set of facilities provided, then, will be evaluated with respect to their support of Archons' primary research goals, rather than with respect to an operationally complete functionality.

Nevertheless, the set of functions described in this document can, and should, describe a set of facilities each of which is functionally closed: i.e. each function provided is complete so that its use can be measured with respect to Archons' research issues. Thus, for example, in the handling of transactions at the client interface, all of the important issues in transaction management are covered, even though other functions, such as file management, may be missing or skimmed. It is not to be assumed that missing or incomplete functionality in the interim tesbed ArchOS represents valueless functionality, but rather that the issues involved with such functions are not within the

[1] Sun Workstation is a trademark of SUN Microsystems, Inc.

primary Archons research interests. Functionality in such areas is expected to be much more complete in later versions of ArchOS incorporating the results of the research performed during this early ArchOS implementation.

This document consists of four chapters. Following this introductory chapter, chapter two will contain a description of the computational model of ArchOS. This will include the client's view of application software structure, with a description of the primary ArchOS client facilities, language issues and the general application software development environment as supported by ArchOS. Chapter three will describe the operating system primitives, including a brief high level description of the underlying operating system activities likely to be undertaken in response to each of the primitives. Particular attention will be given to ArchOS' handling of resource management and consistency, as well as recovery issues. Chapter four will provide a rationale of the design decisions described in Chapters two and three, outlining the tradeoffs made in the selection of application structures and the semantics of the operating system primitives.

4.2 ArchOS Computational Model

In this model, we mix two paradigms in the distributed domain: *object-oriented* and *process-based programming*. While this mixture is hardly novel (e.g., [Lazowska 81]), our paradigm differs from others in several ways. The primary goal is a high level of modularity and maintainability for both the real-time application implementer using ArchOS, and for the ArchOS implementers themselves.

4.2.1 Principal Components

The purpose of ArchOS is to provide an execution environment for a real-time decentralized system, such as a command and control system. The application software of such a system can be considered to be a single distributed program. The distributed program can be described in terms of its primary constituent components, which can be further broken down until the familiar sequential components (processes) are described.

4.2.1.1 Distributed Program

A distributed program consists of one or more arobjects (major program modules -- see Section 4.2.1.2) working toward a single goal. Generally, in a real-time system, the entire system can be considered to be executing a single such distributed program. It is possible, however, that more than one distributed program could be running in a system simultaneously (particularly during system test activities), subject to the availability of resources, but the presence of more than one program may render certain performance specifications untenable because of the associated arbitrarily resolved

resource conflicts. Each program contains a single *Root* arobject which will, in turn, spawn other arobjects.

In fact, because ArchOS will be a testbed operating system for the foreseeable future, it is expected to be used almost exclusively in "system test activities". Because of this, we have planned the client interface to allow more than one program to be executed simultaneously. While it is true that performance specifications may be compromised during such tests, the majority of such tests may not be greatly impacted. ArchOS will keep track internally of which program components are part of each separate program, using a fixed set of policies for handling resource allocation conflicts among the programs. The remainder of this document, however, will concern itself with the handling of a single program.

4.2.1.2 Arobject

An arobject is a distributed abstract data type consisting of two principle parts: a *specification*, and a *body*. An *instance* of an arobject can be dynamically created using the *CreateArobject* primitive. More than one instance of a single arobject can be created, each having a distinct identity, forming an arobject *class*. The arobject identifier returned by the create primitive identifies the particular arobject instance to be addressed during communications, or arobject communications may be initiated to an entire arobject class, using existential or universal quantifiers to specify destination addressing (see description of *Request* primitive in Section 4.3.3.1).

The lifetime of the arobject is under user control and is potentially unlimited. The system build procedures place the uninstantiated copies of the compiled and linked arobjects on long term storage (e.g. disk), but the creation of an arobject causes an instance of the arobject to be created and uniquely named. An arobject instance will be removed at the end of its lifetime, or a kill operation can remove an arobject instance.

4.2.1.2.1 Arobject Specification

The arobject specification, describing the external user's view of the arobject, consists of a set of data types and a set of operations which other arobjects will use to activate services offered by the arobject. The specification, although it completely specifies the external interface to the arobject, makes no commitment with respect to the number of processes within the arobject, their functions, or their distribution. The operations are specified in a manner similar to functions in procedural programming languages: the operation is specified with its name, its input arguments, and its output arguments. All operation invocations and replies use call-by-value semantics.

It is important to note that there is no defined relationship between an arobject's operations and the

entities (processes or procedures) in its body. Any operation could, in principle, be handled by any process in the arobject (see description of the *Accept* primitive in Section 4.3.3.4).

4.2.1.2.2 Arobject Body

The arobject *body* consists of descriptions of private data types, private abstract data types, private operations, private arobjects, and processes. Every arobject body must contain at least one process, but the other components of an arobject body are optional. This private information is visible only to processes within the arobject. If a process is not resident in the same node as an instance of a private abstract data type, the semantics of a call to one of that private abstract data type's procedures are those of a remote procedure call. The semantics of the remote procedure call will ensure that the procedure will be called at most once; it will be the responsibility of the calling process to handle the condition of a procedure never being executed.

Thus, there are a number of potential items in the arobject body, each with a set of semantics controlling their use. The procedures defined within a private abstract data type will determine the access rules to the encapsulated private data, handling mutual exclusion if needed, or providing "dirty" access if this is acceptable to the client system functional specifications. The private data can be any normal data types, but may also be defined as *atomic* or *permanent*. Data defined as atomic will be forced to stable storage upon commitment of a transaction, or restored to its prior, or some equivalent, state upon transaction abort. Permanent data is similar, except that the copying of this data to stable storage is done asynchronously (with respect to the changes made to the data) by the operating system or by an explicit primitive, with no guarantees with respect to consistency. Atomic data access is possible only within transactions (see Section 4.2.4).

Private operations and private data types are exactly the same as their counterparts in the specification part of an arobject, except that their existence is invisible outside the arobject; hence these operations and types may be used only within the arobject. Such operations and types serve to enhance the modularity characteristics of arobjects by providing for nesting of distributed abstract data types. Similarly, private arobjects are simply arobjects which, once instantiated (see *CreateArobject* primitive, Section 4.3.2.1) by an arobject, are visible only to the processes within the outer arobject. Their existence is unknown to external arobjects, and can be used to hide implementation details in exactly the same way as normal abstract data types.

4.2.1.3 Processes

A process is a sequential execution unit within an arobject; it is the dispatchable entity as viewed by the operating system. An arobject contains one or more processes, of which one may be named INITIAL. If such a process exists, it is automatically invoked at arobject creation time (see description of *CreateArobject* primitive in Section 4.3.2.1), providing for initialization of the private variables which comprise the arobject's state. Processes may share a data object with other processes in an arobject by encapsulating it as an instance of a private abstract type. Processes may be explicitly created only by other processes in an arobject: their existence is invisible to processes outside the arobject. Once created, processes are terminated at their own request, at the request of any other process in the arobject, or at the termination of the arobject instance.

Processes are *lightweight*; i.e., no computational state (such as local data) is implicitly transferred to a process via invocation other than its formal input parameters, so the invocation activity can be made to be procedurally inexpensive, particularly if appropriate hardware support is available [Jensen 84]. There is no necessity that all processes in an arobject be located at a single node. The model does not even require that processes sharing variables be located at a single node, but the actual implementation might contain such a limitation.

4.2.1.4 Sample Arobject

The structure of a distributed program and each component of an arobject is illustrated in Figure 4-1. The three small boxes in the left portion of the figure represent three cooperating arobjects. Arobject 2 has made a *Request* of arobject 1, and arobject 1 has sent a corresponding *Reply*. Similarly, arobject 3 has requested some service from arobject 2 and received a reply.

While the small boxes that represent arobjects in Figure 4-1 hint at the internal structure of an arobject (they show that an arobject has two parts and that one part contains internal processes and private abstract data types (padt's)), they do not show much detail. The large box on the right side of the figure is an exploded view of arobject 2. It shows that the two parts of an arobject are a specification and a body, and it shows the various components of each part. (These components were all discussed earlier in this chapter.)

4.2.2 Communication Facilities

Arobjects communicate via invocation parameters and messages. Unlike a remote procedure call (RPC) mechanism, the requestor and server must agree to communicate with each other. Their relationship is thus a symmetrical pair of cooperating arobjects rather than that of a master/slave. Such a symmetrical pair can be used to produce either client/server systems, or cooperating arobject

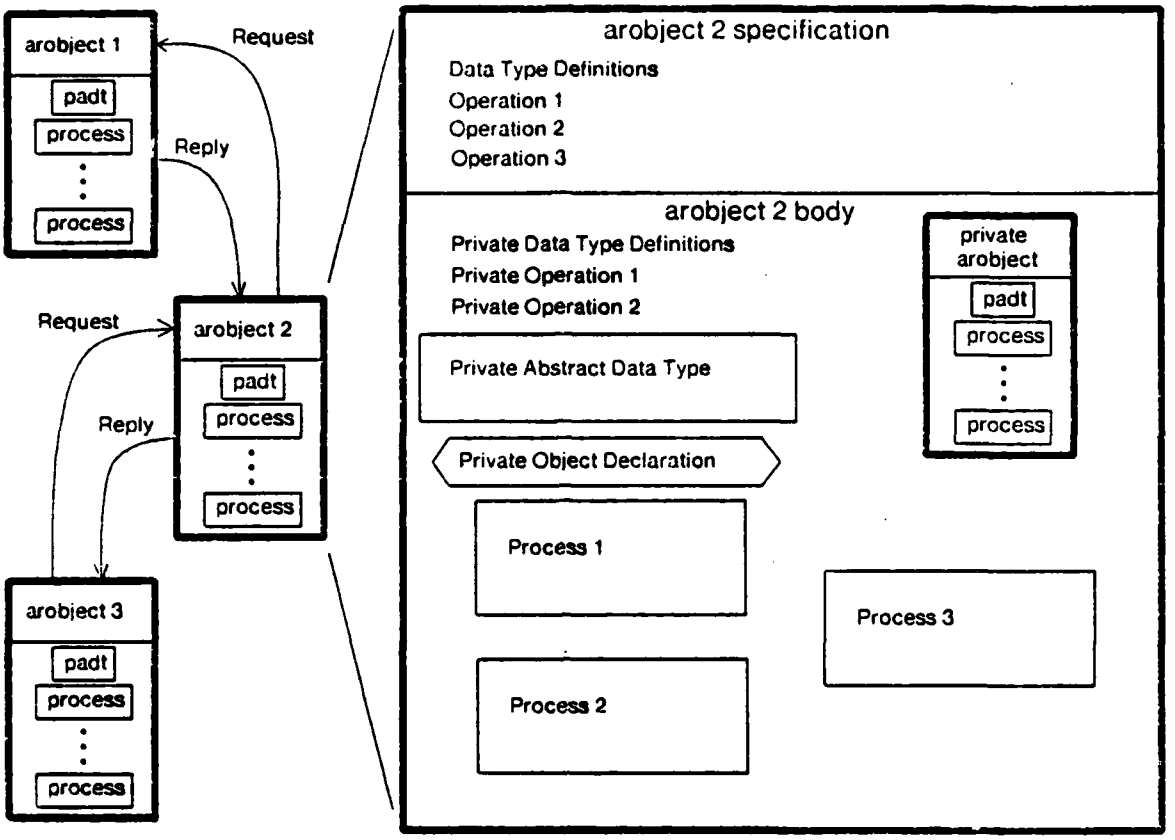


Figure 4-1: Distributed Program using Arobjects

systems, or more likely, a mixture of both. This view pervades both the application and ArchOS implementation paradigms being constructed.

In this example shown in Figure 4-2, the requestor sends a request message explicitly via the *Request* primitive and the server (i.e., Process A) performs an *Accept* primitive. The *Trans-Id* variable in the requestor is set to the unique request transaction identification if the request initiation is successful, and is set to a null transaction id ("NULL-TID") if an error has been detected (e.g., invalid arobject id). The body of the request message must match the corresponding operation parameter template provided in the target arobject specification, using call-by-value semantics. Similarly, the body of the reply message contains the operation's return-parameters. If a transaction is in progress when the request is made, ArchOS will manage the applicable transaction semantics, depending on the transaction type (See Section 4.2.4).

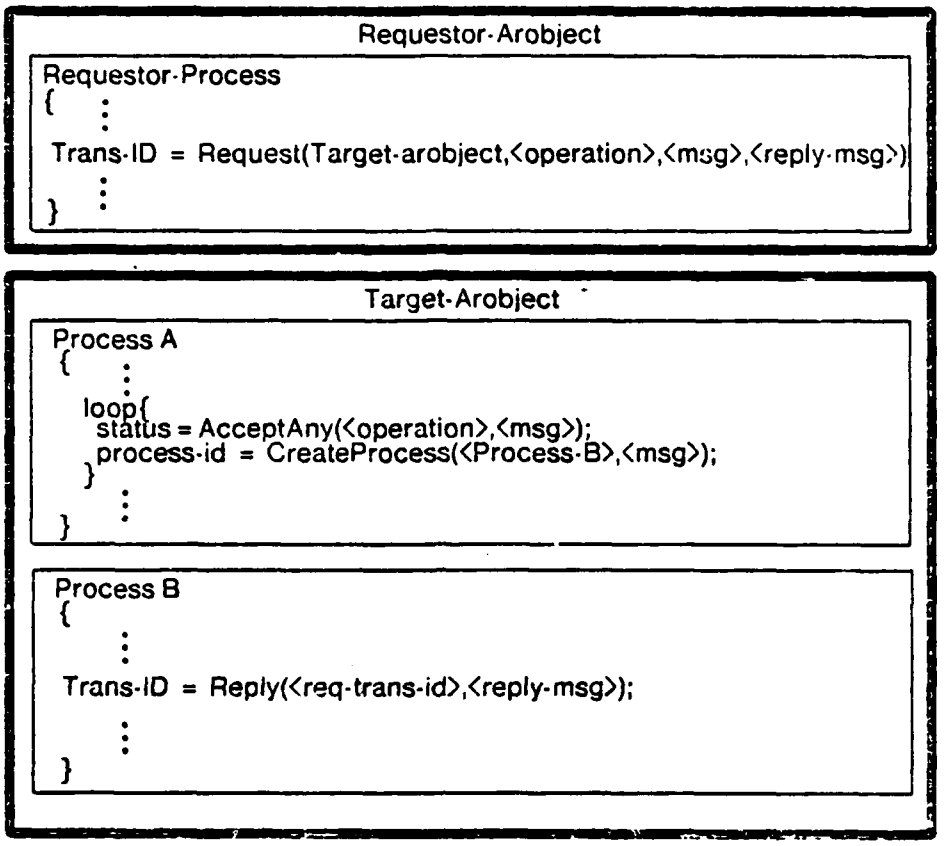


Figure 4-2: Communication Paradigm in ArchOS

4.2.3 System Load and Initialization

Although system generation is beyond the scope of this document, we will assume that a system of application arobjects has been built, and that a directory (possibly partitioned) of the arobjects has also been constructed, and is available to each node. For each distributed program (normally one, but possibly more than one during test phases), one arobject has been identified as a root arobject.

At system startup, once ArchOS has initialized itself and determined its initial state, the directory will be searched for every client root arobject, each of which will then be automatically *Created*, including execution of its INITIAL process. This action will then complete the system load and initialization, with or without operator assistance.

4.2.4 Transactions

ArchOS will use transactions in order to apply the properties that transactions have traditionally displayed in database applications (such as failure atomicity and data permanence) within the operating system. OS-level transactions should facilitate the maintenance of system consistency while simplifying data sharing among multiple processes. In fact, we expect to use transactions extensively within the ArchOS system primitives.

The ArchOS transaction facility does not provide failure atomicity and permanence for all of the data accessed by a transaction. Rather, only the data items that have been explicitly declared to be *atomic* have these properties.

4.2.4.1 Elementary and Compound Transactions

ArchOS will support two types of transactions, elementary transactions and compound transactions, which may be nested in arbitrary combinations. Elementary transactions correspond to traditional nested transactions [Moss 81]. Compound transactions [Sha 84] are supported by ArchOS in order to provide more potential concurrency than elementary transactions and differ from elementary transactions in one important way: when a compound transaction commits, the processing which makes the effects of the transaction permanent and visible (the commit processing) takes place immediately. In this manner, a compound transaction is treated as if it were a top-level transaction, even though it may actually be nested within another transaction.

Compound transactions must be viewed differently than traditional, serializable transactions. In the traditional case, the transaction writer is able to assume that his transaction will be executed as if it were the only transaction in the system. The transaction facility will perform whatever concurrency control is necessary to ensure that the entire transaction will be executed atomically so that no other transaction can view the partial results of this transaction; other transactions can view only the final, committed results. The system thereby ensures that the operations performed on the atomic data objects correspond to some serial ordering of the transactions; this, in turn, guarantees that a set of transactions which individually preserve the consistency of the atomic objects they access will also collectively preserve that consistency.

Compound transactions do not allow the transaction writer to act as though that writer's transaction is the only one active in the system at any given time. Rather, the writer of a compound transaction must realize that, in essense, a compound transaction allows other transactions to view partial results of computations. That is, if a compound transaction is nested within another transaction, when the compound transaction commits, it is implicitly allowing other transactions to view the state of the

atomic data objects that it has manipulated. Since the outer transaction has not yet committed, this state may be considered a partial result of the computation being performed by the outer transaction. (Compound transactions allow partial results to be visible, thus admitting the possibility of side effects, while providing a means of increasing concurrency in applications where such transactions can be employed.)

Due to the fact that a compound transaction can allow other transactions to see partial results of an ongoing computation, compound transactions must be written carefully. In particular, a compound transaction should perform a *consistency preserving* transformation on the set of atomic data objects that it accesses. A consistency preserving transformation is a set of operations that transforms a set of atomic data objects from one consistent state into another consistent state. In this way, no other transaction can ever see a set of atomic data objects in an inconsistent state. For applications in which it is acceptable to allow other transactions to view certain partial results that represent consistent states for a set of atomic data objects, compound transactions may be used, and an increase in overall application and system concurrency can result.

The behavior described above has several important consequences for compound transactions:

- It may be impossible to "undo" (in the sense of Moss's nested transactions) the effects of a committed compound transaction which is nested within another transaction. That is, in the event that a commited compound transaction is later aborted (due to the abortion of a higher level transaction which contains that compound transaction), there is no way that ArchOS can guarantee that the atomic data structures manipulated by that compound transaction can be restored to the same state that they possessed prior to the initiation of the transaction. However, ArchOS does provide a method by which an arobject author can define an operation, called a *compensation operation*, to be associated with each operation defined on a given arobject. It is expected that the arobject writer will write a compensation operation for each arobject operation that could be invoked during a compound transaction. (In some cases, a compensation operation will involve more than simply restoring an object to its original state. For example, after a compound transaction has altered the value of some shared variable, it is possible for other transactions to read, or even change, that value. Later, if compensation must be performed for the committed compound transaction (due to the abortion of a higher-level transaction), it may be necessary to take additional steps to assure that the visibility of the shared variable's value during the interval between the compound transaction's commitment and subsequent abortion has not caused any undesirable side effects. One possible step, for instance, might involve broadcasting a message to all of the potential viewers of the shared variable's value, stating that a transaction abort has caused the value seen most recently to be invalidated.) In the event that one or more arobject operations were invoked during the processing of a committed compound transaction that is subsequently aborted, ArchOS will properly compose the corresponding compensation operations in order to imitate the effects of a traditional "undo" operation. These compensation operations will then transform the states of the atomic data structures manipulated during the compound transaction execution into some member of

a class of states that are *equivalent*, although not necessarily identical, to the pre-transaction states of those data structures.[2]

- Compound transactions can be used to release shared resources before the completion of an entire nested transaction execution, potentially increasing system concurrency.

Transactions are defined by the arobject programmer by means of ArchOS primitives. These primitives appear as programming constructs similar to those used to define *while* and *for* loops. Such a syntactic structure allows a transaction to be placed at any point in any process, while insuring that both the beginning and the end of the transaction occur within a single process.

ArchOS will handle the ordering of compensation operations for aborted compound transactions automatically. This will be done by constructing a sequence of compensation operations corresponding to the operations performed during the execution of a compound transaction. In the event that this compound transaction is committed and subsequently aborted, these compensation operations will be performed in the reverse order of the original execution sequence. As explained above, although ArchOS will initiate the processing of these compensation operations, the operations themselves must be defined by the author of the arobject whose operations were invoked by the compound transaction.

The ArchOS client will often need to access shared data during transaction processing. Such accesses are coordinated by means of a locking mechanism. The client must explicitly request locks on the shared data objects that are needed for the execution of a given transaction. The following section discusses the ArchOS locking facilities in detail.

4.2.4.2 Locks

ArchOS supports two types of locks: *discrete locks* on independent, individual data items, and *tree locks* [Silberschatz 80], which are structures of related discrete locks. In the case of discrete locks, the client obtains (sets) a lock for the desired data item, manipulates the item, and releases the lock.

Tree locks are handled in a somewhat different manner. In this case, there is a tree structure of locks, and there are a few rules that must be followed when accessing the locks. In particular:

- Initially, a client may set a lock located at any point in the tree of locks.
- Subsequently, a client may set any lock whose parent lock is currently held by that client.

---

[2] Strictly speaking, this is not quite true. In fact, the state of the atomic data structures will be equivalent to the state that would have existed had all of the other concurrent committed transactions taken place in the absence of the aborted compound transaction.

- A client cannot acquire a tree lock, release it, and then reacquire it before all of the client's locks in that tree have been released.

- Locks may be released at any time, as long as the above conditions hold.

Tree locks have one important property: if all of the locks involved in a set of computations are contained in a single lock tree and if the above rules are obeyed and if all tree locks are released in a finite amount of time, then deadlocks involving locks in the tree are impossible.

Within transactions, locks are obtained explicitly by the arobject programmer. When a nested elementary transaction commits, its locks are passed to its parent transaction; when a top-level elementary transaction or any compound transaction commits, all of the locks obtained by that transaction are released. (ArchOS will handle this automatically.) When a transaction is initiated from within the scope of a higher-level transaction (see next section for a discussion of transaction scope), it does not automatically inherit the locks which belong to its parent transaction. However, it may attempt to obtain locks held by any of its ancestors by means of the *SetLock* primitive.

In addition to the lock facilities mentioned above, a primitive is provided to release locks. However, the *ReleaseLock* primitive must be used carefully within transactions. Transactions possess desirable properties as a result of the fact that they restrict the freedom with which the locking and unlocking of data items can occur. If a client abuses the locking/unlocking conventions employed by the ArchOS transaction facility by improperly making *ReleaseLock* invocations, then ArchOS may not be able to complete the processing of the client's transaction. Rather, ArchOS will detect the violation of the locking conventions and will terminate the transaction in as orderly a manner as possible. More precisely, ArchOS will abort the transaction upon detection of a lock protocol violation. This may result in an inconsistent or incorrect state for some atomic objects. However, since the transaction can only release locks that it can explicitly name, the integrity of atomic data objects which are manipulated on its behalf can be guaranteed by appropriate use of transactions to encapsulate these other atomic data objects. Such an encapsulation will guarantee that the atomic data objects manipulated by these transactions will be in consistent states and can recover to other consistent states when the lock-violating transaction is aborted. However, since the lock-violating transaction has abused the locking conventions, it may be impossible to properly recover the atomic data objects that it manipulates directly. In that case, ArchOS cannot ensure that these atomic data objects will be in a consistent state. (See the explanation concerning consistency preserving transformations in Section 4.2.4.1.)

Despite the warning given above, the *ReleaseLock* primitive does have "safe" applications. For

instance, it can be used to manipulate tree locks or to release locks that were obtained during non-transaction processing.

4.2.4.3 Transaction Scope, Models, and Lock Passing

This section defines some additional terminology for discussing transactions and introduces the notion of a transaction tree, a structure that allows a simple visualization of the relationships among various transactions. In addition, the rules for lock propagation in the ArchOS transaction facility are specified in more detail.

The preceding discussion concerning transactions dealt with the notion of nested transactions. The transaction tree is a structure that can be used to explain the behavior of a set of transactions. Figure 4-3 contains an example of a transaction tree.

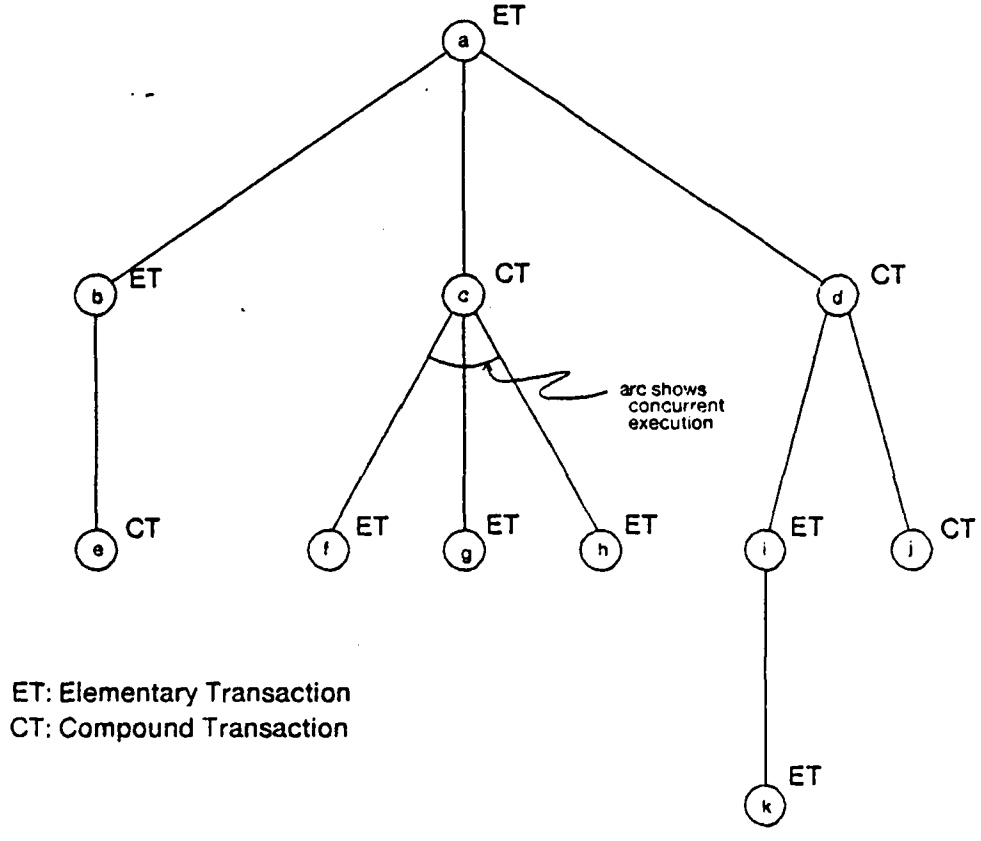


Figure 4-3: Sample Transaction Tree

Each circle in the transaction tree represents a transaction execution. (The transaction nodes have been labeled so that they may be individually referenced later.) Transaction tree nodes that are children of a given transaction tree node represent transactions that are nested within a parent

transaction (that is, either they are contained within the parent transaction's definition or they are executed as a result of invocations performed as part of the parent transaction). Children that are connected to their parent by a single line only are executed in a serial order, with the leftmost child being executed first; children that are connected to their parent by a line through which an arc passes are executed concurrently. (These concurrent executions are typically the result of executing *RequestSingle* or *RequestAll* primitives.)

Each transaction has an associated scope. That scope is delimited by the transaction definition primitive. Any statement that is part of the transaction definition is within the scope of the transaction, as are any statements that are executed as a result of arobject operation invocations or procedure or function calls made by statements within the transaction's scope. With respect to the transaction tree representation, all of the subtree at or beneath the node corresponding to a given transaction lies in the scope of that transaction.

The transaction tree can also be used to explain the ArchOS rules governing lock passing among transactions. As explained in Section 4.2.4.2, the client explicitly obtains the locks that are required by a transaction. When a nested elementary transaction commits, all of the locks that it held are passed to the transaction in which it is nested (its parent in the transaction tree); when a nested compound transaction commits, all of its locks are released.

The rules that determine which locks a transaction may obtain are more involved. Two transactions are said to be *unrelated* if: (1) they are not contained in a single transaction tree, or (2) they are in a single transaction tree and are concurrently executing siblings or descendants of concurrently executing siblings, or (3) they are in a single transaction tree in which one is an ancester of the other and either the descendent is a compound transaction or there is a compound transaction on the path connecting the two transaction nodes in the corresponding transaction tree. (Note that according to this definition, a compound child transaction is always unrelated to its parent transaction.) Two unrelated transactions may compete for locks, and they may hold locks with compatible lock modes for a single data object at any given time. However, if they request incompatible lock modes for a single data object, then one of the competitors will obtain a lock and the other will block until it can receive the desired lock, or it will return to the requestor with an appropriate status indication.

Two transactions are said to be *related* if they are contained in a single transaction tree where one is the descendent of the other, the descendent is an elementary transaction, and there are no compound transactions in the path connecting their respective nodes in the transaction tree. The lock compatibility rules for related transactions are different than those for unrelated transactions. In

this case, the descendant transaction can obtain any lock mode for any lock held by a related ancester in the transaction tree, including incompatible lock modes that would not be allowed if the transactions were unrelated. (Of course, the descendant transaction will have to compete with all of the unrelated transactions in the system in order to successfully obtain the requested lock with the desired mode.)

Examples of both related and unrelated transactions can be found in Figure 4-3. For instance, transaction 'k' is related to transactions 'i' and 'd,' but it is unrelated to transaction 'a' (due to point (3) in the above definition of unrelated transactions). Also, while transactions 'f,' 'g,' and 'h' are all related to transaction 'c,' they are all unrelated to one another (due to point (2) in the above definition of unrelated transactions). Finally, there are only two other related transaction pairs in the figure: transaction 'i' is related to transaction 'd' and transaction 'b' is related to transaction 'a.'

4.2.5 Real-Time Facilities

As previously stated, ArchOS is a real-time operating system; for ArchOS, this statement carries a number of important implications. In this work, we define a real-time operating system to be one which manages its system resources to meet user-defined deadlines. Processes will be scheduled using *deadline-driven scheduling* with reference to user-defined policies (See Section 4.2.6). This means that when the system determines that processing resources are sufficient to meet user deadlines at each node, it will meet them, but when resources are insufficient, user policy will guide the operating system in its decisions as to which deadlines should be missed or whether some processes should be relocated.

Examples of user policies to be implemented in the event of insufficient resources include:

- Minimize average lateness
- Minimize maximum lateness
- Minimize number of late processes
- Minimize priorities of late processes

In scheduling terminology, these policies actually define objective functions for the resulting scheduling algorithm(s). Scheduling techniques for some of these objective functions (e.g., minimize maximum lateness) are well known, while for most others optimum algorithms are known to be intractable. ArchOS will use best effort decision making to implement policies such as these and many other similar user policies as closely as possible.

In order to handle real-time constraints, each ArchOS primitive must have its execution time bounded. This bound may be probabilistic, and should be determined with respect to the significant events or actions involved in fulfilling the request (e.g., maximum number of internode messages, maximum cpu time used).

The local scheduling model used by each ArchOS node to manage its real-time load consists of a set of n active processes p_i, i between 1 and n. For each p_i, a stochastic execution time C_i and a deadline T_i are known. C_i is a value estimated by the process' implementer and measured by the system during execution. As a part of the set of user controllable policies described above for handling missed deadlines, a value function V_i will be defined for each processor to determine the value to the system associated with achieving a particular degree of lateness for process p_i. The value functions themselves are either chosen by ArchOS with reference to the user policies, or may be provided directly by the user policies, thus creating an extremely large set of potential overload policies.

The system will continuously monitor its performance with respect to the likelihood of missing currently known deadlines, using a best effort to arrange scheduling to distribute the lateness should missed deadlines occur. This processing is currently a critical portion of the Archons research effort, and the results of this research will be directly applied to this scheduling.

During process scheduling at a single node, an overload condition may be detected in which deadlines may be missed. The scheduling algorithms, under control of the appropriate policies, may determine a (sub)set of processes which should be removed from the local node and moved elsewhere. The decision of where they should be moved will be made by an algorithm to be developed, also under control of the application defined policy.

Primitives are available for specification of periodic process execution, user specified delay, real-time clock management, and lateness doctrine policy specification. With respect to the ArchOS primitives, deadlines are defined by adding the client-defined deadline interval (see *Delay* and *Alarm* primitives, Section 4.3.9) to the request time, which is defined to be: (1) the scheduled periodic process execution time, (2) the expiration time of a *Delay* or *Alarm* primitive currently in progress, (3) the time at which a new process becomes ready for execution following a *CreateProcess* primitive. Processes for which deadlines have not been defined, if any, will be scheduled with the objective of maximum throughput subject to the constraint that deadlines will first be met for processes with deadlines.

4.2.6 Policy Definitions

ArchOS exists to support a distributed (application) program. In order to provide a flexible environment that the program writer can tailor to satisfy specific application-dependent requirements, ArchOS allows the client to define the policies used to manage certain system resources.

In general, the ArchOS client may dynamically specify the policy to be followed with respect to the management of a specific resource. For instance, a process scheduling policy may specify the manner in which processor time is managed. In order to support a wide range of alternative policies, some of which may be defined by the client, ArchOS will include a set of mechanisms that will be combined as appropriate to supply the foundation for the facility that the client has selected.

The Archons researchers are interested in studying the specification of policy by the application programmer, as well as the separation of policy and mechanism within a given resource management facility. In ArchOS, the integration of client-selected policies into the system will be studied in two areas: process scheduling policies and process reconfiguration policies. Process scheduling policies are discussed at some length in this section to provide examples for the ArchOS policy definition facility since these ideas have been actively pursued by Archons researchers; process reconfiguration policies, which govern the dynamic migration of processes from node to node, are not as well developed and are an area of future research.

Consider the definition of a process scheduling policy by the client. There are two different situations under which processes are scheduled: the situation in which there are sufficient processing resources to satisfy all of the client-specified service requests, and the situation in which the demand for processing resources is greater than their supply. At this time ArchOS policy management is only concerned with the latter case, and this is reflected in the support provided for client-specified scheduling policies.

In the case where the demand for processing resources is greater than their supply, the client has two options: the client may issue a general policy statement which corresponds to a pre-defined scheduling policy, or the client may define a new policy which adheres to the general model that the mechanisms underlying the ArchOS process scheduling facility support. These mechanisms are aimed at optimizing the value of a function, known as the *objective function*. In ArchOS, the objective function will maximize the sum of a *value function* evaluated for each runnable process on a given node. Whenever the client selects a pre-defined scheduling policy, ArchOS will transform this selection into the appropriate value (and hence, objective) function. In situations where the client wants to use a scheduling policy that ArchOS does not "know about," the client must explicitly

specify the value function to be used. (The client accomplishs this by means of specifying the parameters for a value function. Value functions are also discussed in Section 4.2.5.)

The set of pre-defined scheduling policies which may be selected when the demand for processing resources exceeds the supply includes the following examples (See Section 4.2.5 for a discussion of process deadlines and real-time considerations, in general.):

- minimizing the number of deadlines missed;
- minimizing the average lateness;
- minimizing the maximum lateness;
- maximizing the minimum lateness;
- minimizing the weighted tardiness. (Tardiness is defined to be non-negative lateness, and the weighting factor can incorporate process or arobject priorities into the scheduling computation.)

The implementation of client policy definition in ArchOS will include the notions of *policy names* and a corresponding set of *policy modules*. For example, *Schedule* and *Reconfigure* will be recognized by the system as *policy names* and their bodies can be kept as a separate *policy module*. All of the client-defined policy modules are maintained by a *policy set arobject*, which will be instantiated by the application program. Thus, it will also be possible to add, delete, or change the existing policy module for a given policy by invoking a corresponding operation of the policy set arobject during run time.

Each policy module can be built by referring to a set of *policy attributes*. These attributes are used to determine the policy to be carried out by ArchOS or to pass information to the policy module so that it can make a particular policy decision. For instance, the arobject priority and lateness doctrine parameters listed below could be used by ArchOS to translate the client's policy desires into an objective function to handle process scheduling.

Inclusion of a policy set arobject in an application system will result in its INITIAL process being scheduled during system initialization. This arobject may then define a set of policy attributes for this program. Some examples of attributes that might be used for the *Schedule* or *Reconfigure* policies are

- Arobject priority -- relative scheduling priority of instances of one arobject versus instances of other arobjects in the application program during periods when there are not enough computing resources to satisfy all demands. It should be noted that in the event

of more than one application program running on the ArchOS system, no relative priorities may be determined between them, so performance of either or both may be adversely affected. (As stated in Section 4.2.1.1, ArchOS is designed expressly to support a single program, or set of arobjects. This set of arobjects may well perform computations that would normally be associated with a number of separate "tasks." However, as far as ArchOS is concerned, this collection of tasks comprises a single distributed program since all of the relative priorities among the component arobjects are defined.)

- Lateness doctrine -- rules to be followed in the event that deadlines must be missed. This doctrine indicates the nature of processing which will characterize degraded modes, including such possibilities as maximizing the minimum lateness, minimizing the maximum lateness, minimizing the average lateness, etc. Execution of these doctrines will be done on a best effort basis utilizing the available information, even though incomplete or inaccurate, maximizing its value, and making the decision as near optimal as possible, consistent with the application performance requirements.

4.3 ArchOS Primitives

ArchOS primitives are defined as a set on operations which manipulate various system and kernel arobjects. The primitives can be classified as *system* and *kernel* primitives. The system primitives are provided by system arobjects which exist above the kernel while the kernel primitives are defined within a set of kernel arobjects. In this section, we will specify all of the ArchOS primitives (both kernel and system) which are visible to a client. Other kernel primitives will be described in the ArchOS System Architecture Specification.

4.3.1 Specification of Primitives

We will use the following format to describe the specification of ArchOS primitives. The specification of a primitive consists of two parts. The first part describes the functionality of the primitive, and the second part defines the actual interface for a client process by using a syntactic template shown in below.

It should be noted that since we adopted a C-like syntax in this document, many style of type declataions are adapted from C [Kernighan 78]. For instance, a pointer called "ptr" to a type, say "$TYPE_X$", will be specified as "$TYPE_X$ *ptr". An optional argument, say arg_0, of a primitive will be indicated by [, arg_0] in its argument list.

---

val_0 = **Primitive**(arg_1, ..., arg_k)

$TYPE_0$ val_0 This indicates that the "val_0" has the type "$TYPE_0$". It also includes the type definition of $TYPE_0$ if necessary.

$TYPE_1$ arg_1 This line specifies the type of $argument_1$.

...

$TYPE_k$ arg_k This line specifies the type of $argument_k$.

The first line in this template shows that an ArchOS primitive, called **Primitive** requires *k* arguments, such as arg_1, ..., arg_k, and returns a value.

The additonal part may consists of error or abnormal conditions related to the above primitive invocation and may have the following explanations.

On Error: This part explains the types of errors that can occur and what values will be returned in response to an error. In general, a client can get detailed error information by looking at a specific area called an *error block*. The error block must be declared by using a *SetErrorBlock* primitive (see Section 4.3.11.5).

On Timeout: If a primitive has a timeout argument, then this may explain what happens after a timeout occurs.

This may also contain extra comments such as follows.

Note: This is an example of additional comments on this primitive.

4.3.2 Arobject/Process Management

The arobject and process management provides creation and destruction of arobject instances as well as process instances. The *CreateArobject* and *CreateProcess* primitives create an instance of an arobject and a process respectively. Similary, the *KillArobject* and *KillProcess* primitives kill a specific instance.

The lifetime of an arobject instance can vary, is determined by the lifetime of the last active process instance within that arobject instance. In other words, if an instance of an arobject is killed, all of its internal processes will be halted and removed. A process may terminate by normal exit (i.e., reaching the end of its body) or by an explicit *KillProcess* primitive.

4.3.2.1 Create

A *CreateArobject* primitive creates a new instance of an arobject at an arbitrary node or a specified node. Similarly, a *CreateProcess* primitive creates a new instance of a process in the arobject. The selection of a node is made automatically by ArchOS unless overridden by the *Create* operation. Upon arobject creation, the arobject's INITIAL process is automatically dispatched.

An optional set of parameters can be passed to the INITIAL process when the arobject is instantiated by using an initial message (i.e., "init-msg").

---

arobject-id = **CreateArobject**(arobj-name [, init-msg] [, node-id])
process-id = **CreateProcess**(process-name [, init-msg] [, node-id])

AID arobject-id — The unique identification of the instantiated arobject.

PID process-id — The unique identification of the instantiated process.

AROBJ-NAME arobj-name
The name of arobject to be instantiated.

PROCESS-NAME process-name
The name of process to be instantiated.

MESSAGE *init-msg
A pointer to the initial message which contains initial parameters for the INITIAL process.

NODE-ID node-id — Node identification (optional). An actual node may be designated, or a node selection criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).

On Error: If a *CreateArobject* or *CreateProcess* primitive fails, a "NULL-AID" or "NULL-PID" will be returned, respectively. The detailed error code can be found in an error block which is defined by issuing a *SetErrorBlock* primitive. (see Section 4.3.11.5).

4.3.2.2 Kill

The *KillArobject* and *KillProcess* primitives remove a process and arobject instance respectively. An arobject may be killed only by one of its own processes (suicide allowed, no murder). In order to kill another arobject, the target arobject must have an appropriate operation defined within its specification so it can kill itself.

A process can be killed only by a process which exists in the same arobject instance.

---

val = **KillArobject**(aid)
val = **KillProcess**(pid)

BOOLEAN val TRUE if this killing was successful; otherwise FALSE.

AID aid The arobject id of the target arobject to be killed.

PID pid The process id of the target process to be killed.

On Error: If the specified arobject does not exist in the system, or a target process does not exist in the requestor's arobject instance, a "FALSE" value will be returned.

4.3.2.3 SelfID and ParentID

The *SelfAid* primitive returns the requestor's arobject id and the *ParentAid* primitive returns the parent's arobject id of the specified arobject. The *SelfPid* primitive returns the process id (pid) of the requestor and the *ParentPid* primitive returns the parent's pid of the the specified process.

---

aid = **SelfAid()**
paid = **ParentAid**(aid-x)
pid = **SelfPid()**
ppid = **ParentPid**(pid-x)

AID aid, aid-x, paid The arobject id.

PID pid, pid-x, ppid The process id.

On Error. If a non-existing aid or pid is given to *ParentAid/Pid*, a "NULL-AID" or "NULL-PID" will be returned.

4.3.2.4 BindName

Any arobject or process can have a (run time) reference name defined by these binding primitives within a single distributed program. The *BindArobjectName* and *BindProcessName* primitives bind the requested instance of an arobject or process to a reference name. Unless an *Unbind* primitive is executed, the lifetime of a binding is the same as the lifetime of an arobject or process instance.

This binding allows an arobject or a process to have more than one reference name, or a single reference name can be bound multiple arobject or process instances.

To cancel the current binding, a process must use the appropriate unbind primitive.

---

val = **BindArobjectName**(aid, arobj-refname)
val = **BindProcessName**(pid, process-refname)

BOOLEAN val — TRUE if this binding was successful.

AID aid — The arobject id.

PID pid — The process id.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid.

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid.

On Error: A "FALSE" value will be returned in the case of an error. For instance, if a client attempts to bind non-existant arobject or process instance to a reference name, a "FALSE" value will be returned.

4.3.2.5 UnbindName

The *UnbindArobjectName* and *UnbindProcessName* primitives release the current binding between the specified instance of an arobject or process and a reference name.

---

val = **UnbindArobjectName**(aid, arobj-refname)
val = **UnbindProcessName**(pid, process-refname)

BOOLEAN val — TRUE if this unbinding was successful; otherwise FALSE.

AID aid — The arobject id.

PID pid — The process id.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid.

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid.

On Error: A "FALSE" value will be returned, if a client attempts to release the binding for a non-existing reference name.

4.3.2.6 FindID and FindAllID

A *FindID* primitive returns the unique id (i.e., aid or pid) of the given arobject or process in a specific search domain. A search domain can be specified with respect to all of the internal arobjects, external arobjects, a local node, a remote node, or a reasonable combination of among four. If more than one instance uses the same reference name, the unique id of any one of them will be returned. A *FindAllID* primitive, on the other hand, returns all of the aid's and pid's which correspond to the given reference name.

---

aid = **FindAid**(arobj-refname [, preference])
pid = **FindPid**(process-refname [, preference])
aid-list = **FindAllAid**(arobj-refname [, preference])
pid-list = **FindAllPid**(process-refname [, preference])

AID aid — The arobject id.

PID pid — The process id.

AID-LIST aid-list — The list of corresponding aid's.

PID-LIST pid-list — The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

PREFERENCE preference
The preference can specify a search domain such as "INTERNAL", "EXTERNAL", "LOCAL", "REMOTE", "INTERNAL-LOCAL", "INTERNAL-REMOTE", "EXTERNAL-LOCAL", "EXTERNAL-REMOTE".

On Error: If the FindAid or FindPid primitive fails, a "NULL-AID" or "NULL-PID" will be returned respectively. If a FindAll primitive fails, a "NULL-AID-LIST" or "NULL-PID-LIST" will be returned.

4.3.3 Communication Management

The communication management provides an inter- and intra-node communication facility among cooperating arobjects. A process can invoke an operation at a specific instance of an arobject by sending a request message or can invoke the same operation at multiple instances of an arobject which have been bound to a single reference name. In the later case, the multiple computations are performed concurrently.

In particular, a process can send any arobject instance a request message to invoke an operation without knowing its actual location. A process can also invoke a private operation which is defined within its local arobject.

There are essentially three types of primitives to provide flexible cooperation among arobjects: *Request*, *Accept*, and *Reply*. In addition to these, the *RequestAll*, *RequestSingle*, and *GetReply* primitives are added in order to interact with multiple instances of arobjects concurrently. All of the communication primitives are executed as transactions so that ArchOS can provide properties such as failure atomicity and permanence. (see Section 4.3.6).

A message consists of a header and a body. The message header will be generated by ArchOS and contains control information. The message body carries all of the parameters and will be set by the client process. Since each arobject has a separate address space, parameters must be sent using call-by-value semantics.

4.3.3.1 Request

The *Request* primitive provides remote procedure call semantics in which the requesting process invokes an operation by sending a message and blocks until the receiving arobject returns a reply message. Identically, if the receiver arobject is the same arobject, a local operation will be invoked. When the reply is generated, it is sent to the requestor which is then unblocked. If the requesting process needs to limit the allowed response time, it may do so by creating a small process to handle the timeout condition.

---

trans-id = **Request**(arobj-id, opr, msg, reply-msg)

TRANSACTION-ID trans-id
The transaction id of the transaction on whose behalf the request is being made.

AID arobj-id The unique id of the receiving arobject.

OPE-SELECTOR opr
The name of the operation to be performed.

MESSAGE *msg A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call-by-value semantics must be used).

REPLY-MSG *reply-msg
A pointer to the reply message.

On Error: The *Request* primitive may fail in the following situations:

- The destination arobject's node is not available.
- The destination arobject does not exist.
- The target operation is not defined.
- The request message does not match the receiver's message type.

If the *Request* primitive fails, a "NULL-TID" will be returned.

4.3.3.2 RequestSingle and RequestAll

The *RequestSingle* and *RequestAll* primitives can send a request message and proceed without waiting for a reply message. The *RequestSingle* primitive provides nonblocking one-to-one communication and, the *RequestAll* primitive supports one-to-many communication. The requesting process may thus invoke an operation on more than one instance of an arobject or process with one request. To receive all of the replies, the *GetReply* primitive may be repeated until a reply with a null body is received.

---

trans-id = **RequestSingle**(arobj-id, opr, msg)
trans-id = **RequestAll**(arobj-refname, opr, msg)

TRANSACTION-ID trans-id
The transaction ID of the transaction on whose behalf the request is being made.

AID arobj-id The ID of the receiving arobject.

AROBJ-REFNAME arobj-refname
The reference name of the receiving arobject(s).

OPE-SELECTOR opr
The name of the operation to be performed.

MESSAGE *msg A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call by value semantics must be used).

On Error: The *RequestSingle* and *RequestAll* primitives fail if similar to those situations mentioned in the previous section happen. If the *RequestSingle* or *RequestAll* primitive fails, a "NULL-TID" will be returned.

4.3.3.3 GetReply

The *GetReply* primitive receives a reply message which has the specific transaction id generated by the preceding *RequestAll* primitive. If the specific reply message is not available, then the caller will be blocked until the message becomes available.

---

aid = **GetReply**(req-trans-id, reply-msg)

AID aid — The arobject id of the replying arobject.

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

REPLY-MSG *reply-msg
A pointer to the reply message.

On Error: The GetReply primitive fails, if the specified transaction does not exist; a "NULL-AID" will then be returned.

4.3.3.4 Accept and AcceptAny

The process responsible for an arobject operation receives a message using the *Accept* primitive. Using the selection criteria specified, the operating system selects an eligible message from the arobject's input queue and returns it. The process operates on the message, responding with a reply when processing has been completed. If no suitable message is in the request message queue, then the caller will block until such a message becomes available.

The *AcceptAny* primitive can receive a message from any arobject instance with any operation (i.e., "ANYOPR") or a specified operation request. The primitive can return the requestor's transaction id, specified operator, and requestor's aid. The *Accept* primitive can receive a message from a specific requestor arobject and returns the requestor's transaction id and the requested operator.

(req-trans-id, req-opr, requestor) = **AcceptAny**(opr, msg)

(req-trans-id, req-opr) = **Accept**(requestor, opr, msg)

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr, req-opr
The name of operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the transaction on whose behalf the request is made.

MESSAGE *msg A pointer to the message buffer.

On Error: The *AcceptAny* primitive fails if the specified operation does not exist. Similarly, the *Accept* primitive fails if the requestor or the operation does not exist. If the *AcceptAny* or *Accept* primitive fails, a "NULL-TID", "NULL-OPR", and "NULL-AID" will be returned.

4.3.3.5 CheckMessageQ

The CheckMessageQ primitive examines the current status of an incoming message queue without blocking the caller process. The primitive must specify a message queue type, either "request-queue" or "reply-queue". The request-queue queues all of the non-accepted request messages and is allocated for each arobject instance. The reply-queue maintains all of the non-read reply messages and is assigned to every process instance.

A message can be selected based on the sender's arobject id, operation name, and/or transaction id. If more than one argument is given, only messages which satisfy all of the conditions will be returned. If no corresponding message exists in a specified message queue, a "NULL-POINTER" will be returned.

---

ptr-mds = **CheckMessageQ**(qtype, requestor, opr, req-trans-id)

MSG-DESCRIPTORS *prt-mds
Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype This indicates either "request-" or "reply-" message queue.

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr
The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

On Error: The *CheckMessageQ* primitive fails if the specified message queue, operation, or the transaction id does not exist. If the *CheckMessageQ* primitive fails, a "NULL-POINTER" will be returned.

4.3.3.6 Reply

After processing an *Accept* primitive, a process must use a *Reply* primitive to send the completion message to the requesting process. At the requestor's site, the completion message is received by the second half of the synchrouns *Request* primitive or the *GetReply* primitive. It should be noted that the reply need not necessarily be sent from the same process which accepted the operation. In other words, the requestor's transaction id is used to determin a proper reply message.

---

trans-id = **Reply**(req-trans-id, reply-msg)

TRANSACTION-ID trans-id
The transaction id of this *Reply* primitive (not the requestor's transaction id)

TRANSACTION-ID req-trans-id
The transaction id of the request that has been serviced.

MESSAGE *reply-msg
A pointer to the reply message.

On Error: The *Reply* primitive fails if the specified transaction has already been aborted. If the *Reply* primitive fails, a "NULL-TID" will be returned.

4.3.4 Private Object Management

A process can dynamically create a private object, which is an instance of a private abstract data type defined in its arobject, at any node. If a private abstract data type has instances of atomic or permanent data objects, the actual data objects will be allocated in non-volatile memory

4.3.4.1 Allocate/Free Object

An *AllocateObject* primitive allocates an instance of a private abstract data type at any node and a *FreeObject* primitive deallocates the specified instance.

---

object-ptr = **AllocateObject**(object-type, parameters [, node-id])

val = **FreeObject**(object-ptr)

OBJECT-PTR object-ptr
A pointer to the allocated private data object.

OBJECT-TYPE object-type
The object-type indicates the name of a private abstract data type.

BOOLEAN val TRUE if the object was released successful; otherwise FALSE.

NODE-ID node-id Node identification. An actual node may be designated, or a node selection criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).

On Error: If the object-type is not defined an *AllocateObject* primitive fails and returns a "NULL-POINTER". If the object-ptr is not pointing to a proper permanent object, then the *FreeObject* primitive fails and returns "FALSE".

4.3.4.2 FlushPermanent

A *FlushPermanent* primitive blocks the caller until the specified data object is saved in non-volatile storage.

FlushPermanent(object-ptr, size)

OBJECT-PTR object-ptr
A pointer to the permanent data object.

INT size The number of bytes which must be flushed into permanent storage.

On Error: The *FlushPermanent* primitive fails if the specified object does not exist.

4.3.5 Synchronization

ArchOS provides two levels of synchronization facilities. One, the critical region, is for controlling the concurrent access to a single shared object within an arobject, while the other, the locak, is for controlling accesses from concurrent transactions.

The critical region scheme should be used when the shared object does not need to provide failure atomicity. That is, a client may be able to access an inconsistent state of the object. On the other hand, if these objects are atomic objects and are accessed from transactions, then a client must be allowed to see only consistent objects. In the critical region scheme, mutual exclusion is achieved by implicit locking by using an event variable, while the transactions require an explicit lock on the atomic object. The *CreateLock* and *DeleteLock* primitives are provided to create and remove a lock for the explicit locking scheme. Actual locks can be set by using a *SetLock* or *TestandSetLock* primitive and can be released by using a *ReleaseLock* primitive.

4.3.5.1 Region

The *Region* construct provides a simple mutual exclusion mechanism for controlling concurrent accesses to shared objects. A timeout value must be specified in order to bound the total execution time in the critical region including any time spent waiting to enter the region. If a timeout occurs, a client process is forced to exit from the critical region and an error status is returned.

Region(ev, timeout){ . . . }

EVENT-VAR ev — The event variable consists of a waiting queue of client processes and an event counter.

TIMEOUT timeout — The timeout value should indicate the maximum execution time for this critical region including the waiting time.

On Timeout: A client should be able to check whether the critical region was exited due to a timeout by checking error information in its error block (see Section 4.3.11.5).

4.3.5.2 CreateLock and DeleteLock

The *CreateLock* primitive creates either a *tree-type* or *discrete-type* lock and returns a unique *lock id*. Since a lock id is not bound to any data object explicitly, a client must be responsible for utilizing the lock in accordance with the locking protocol. The *DeleteLock* primitive removes the specified lock from the system.

---

newlock-id = **CreateLock**([parent-lockid])
val = **DeleteLock**(lockid)

LOCK-ID newlock-id
A new lock id will be returned.

BOOLEAN val TRUE if the lockid is removed successfully; otherwise FALSE.

LOCK-ID parent-lockid
If the created lock must be a tree-type lock, then its parent-lockid must be specified. If the parent-lockid is a "NULL-LOCK-ID", then the new lock will be the root of a new lock tree. If a parent-lockid is not given, then the new lock will be a discrete-type lock.

LOCK-ID lockid The lockid to be deleted. If the lockid is a tree-type lock, then the entire subtree of which this lock is the root will be deleted.

On Error: If a parent lockid does not exist, then the *CreateLock* primitive fails and returns a "NULL-LOCK-ID". A *DeleteLock* primitive fails if the specified lockid does not exist, and returns "FALSE".

4.3.5.3 SetLock, TestLock, and ReleaseLock

The *SetLock* primitive sets a "tree-type" or "discrete-type" lock on arbitrary objects by specifying a lock key and its mode. If a requested lock is being held, the caller will block until it is released. The *TestandSetLock* primitive also tries to set a lock, however, it will return a "FALSE" if the lock is being held. If the request lock is a tree-lock type, then the *SetLock* and *TestandSetLock* primitives may also fail due to the violation of the tree-lock convention (See Section 4.2.4.2).

The *TestLock* primitive checks the availability of a specified lock with a lock mode. In the case of a tree lock, it also checks whether the locking would be legal in the corresponding lock tree. The *ReleaseLock* primitive can release the lock on an object which was gained by the *SetLock* or *TestandSetLock* primitive explicitly.

---

sval = **SetLock**(lock-type, lockid, lock-mode)
sval = **TestandSetLock**(lock-type, lockid, lock-mode)
tval = **TestLock**(lock-type, lockid, lock-mode)
rval = **ReleaseLock**(lock-type, lockid, lock-mode)

INT sval 1 if the specified lock is set; 0 if the lock is not set. A negative value will be returned if an error occured.

INT tval — 1 if the specified lock is being held; 0 if the lock is not being held. A negative value will be returned if an error occured.

INT rval — 1 if the specified lock is released; 0 if the lock is not released. A negative value will be returned if an error occured.

LOCK-TYPE lock-type
The lock type can be either "TREE" or "DISCRETE".

LOCK-ID lockid — The lockid indicates the unique id of a lock.

LOCK-MODE lock-mode
The lock mode can be "READ", "WRITE", etc.

On Error: An error may occur if a non-existent lock id or lock mode is used for the above primitives. A detailed error condition, such as a non-existent lock or lock mode, is available by looking at the caller's error block (see Section 4.3.11.5).

4.3.6 Transaction/Recovery Management

ArchOS can allow a client process to create a compound transaction or an elementary transaction which can be nested in any combination. By using nested elementary transactions, a client can use a traditional "nested transactions" mechanism. In addition to this, the compound transaction provides a mechanism which can commit the transactions at the end of the current scope without delaying the commit point to end of the top-level transaction. Since a completed nested compound transaction cannot be undone, ArchOS provides a mechanism to perform corresponding compensate actions automatically (see Section 4.2.4).

It should be noted that ArchOS cannot generate such a sequence of compensate actions automatically. However, ArchOS provides a mechanism to execute the defined compensate actions in the proper sequence to make the status of each affected atomic object into a member of an *equivalence class* of its correct "pre-execution" state. (See Section 4.2.4.1)

4.3.6.1 Compound Transaction

A compound transaction construct creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

When a compound transaction starts, no locks will be inherited from its parent transaction if one exists. That is, all of its locks must be obtained within this scope by means of the *SetLock* primitives.

(See Section 4.3.5.3). However, at the end of the compound transaction scope, ArchOS releases all of the locks for this transaction automatically. It is also possible to release locks before the end of the transaction scope by using a *ReleaseLock* primitive explicitly.

If a compound transaction must abort, an *AbortTransaction* primitive (see Section 4.3.6.4) performs the necessary compensate actions, breaks the current transaction scope, and passes control to the end of the transaction scope.

CT(timeout){ ... <transaction steps> ... }

TIME timeout — The timeout value indicates the maximum lifetime of this compound transaction

<transaction steps> Atomic objects can only be accessed and altered, within these transaction steps.

On Timeout: The current transaction and all of its child transactions will be aborted. That is, ArchOS will execute all of the necessary compensate actions and undo. After completion of these actions, the status of atomic objects should be consistent and be one of the member of the *equivalence class* of its pre-(transaction) execution state.

4.3.6.2 Elementary Transaction

An elementary transaction construct also creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

When an elementary transaction starts, it can obtain its ancestor transaction's locks if there is an ancestor. That is, this elementary transaction may access atomic objects which were manipulated by the higher level transactions. If the ancestor has modified atomic objects, these modifications will be visible to this transaction. At the end of an elementary transaction scope, ArchOS will propagate all of its locks to the parent transaction if one exists. If the elementary transaction is the topmost transaction, then all of its locks will be released at this point and all of its atomic objects will be committed.

If an elementary transaction must abort, an *AbortTransaction* primitive (see Section 4.3.6.4) performs the necessary "undo" actions, breaks the current transaction scope, and passes the current control to the end of the transaction scope.

ET(timeout){ . . . <transaction steps> . . . }

TIME timeout The timeout value indicates the maximum lifetime of this elementary transaction.

<transaction steps> Atomic objects can only be accessed and altered, within these transaction steps.

On Timeout: The current transaction and all of its child transactions will be aborted. That is, ArchOS will execute all of the necessary undo and compensate actions automatically. After completion of these actions, the status of all affected atomic objects should be consistent and be "identical" to the correct initial (pre-execution) states.

4.3.6.3 SelfTid and ParentTid

The *SelfTid* primitive returns the id of the current transaction and the *ParentTid* primitive returns the parent transaction id of the given transaction id.

---

mytid = **SelfTid**()
ptid = **ParentTid**(tid)

TID mytid *The id of the current transaction.*

TID tid The id of the specific transaction.

TID ptid The parent's tid of the given transaction "tid".

On Error: If the *ParentTid* primitive fails, a "NULLTID" will be returned.

4.3.6.4 AbortTransaction

The *AbortTransaction* primitive aborts the specified transaction and all of its child transactions within the same transaction tree (See Figure 4-3 in Section 4.2.4.3). If the transaction that invokes the *AbortTransaction* primitive does not belong to same the transaction tree as the transaction which is to be aborted, a client cannot abort that transaction. This primitive executes all of the necessary "undo" or "compensate" actions, based on the transaction type, and breaks the current transaction scope. After completion of these actions, the status of all affected atomic objects will be consistent and returned to either "identical" to or "a member of the equivalence class" of their initial (pre-execution) states.

The *AbortIncompleteTransaction* primitive also aborts all of the outstanding incomplete transactions which had been initiated by an outstanding *RequestSingle* or *RequestAll* primitive. In other words, all of the nested transactions which belong to the specified request transaction but have not yet completed (committed) will be aborted.

---

val = **AbortTransaction**(tid)
val = **AbortIncompleteTransaction**(req-tid)

BOOLEAN val — TRUE if the transaction was aborted successfully; otherwise FALSE.

TID tid — The id of the transaction.

TID req-tid — The transaction id of the *RequestSingle* or *RequestAll* primitive.

On Error: If an *Abort* primitive is called from a transcation which is not a parent of, or identical to, the designated transaction, the primitive will fail and return FALSE.

4.3.6.5 TransactionType

The *TransactionType* primitive returns the type of the given transaction (a compound or elementary) and also indicates the transaction level.

---

trantype = **TransactionType**(tid)

TRANTYPE trantype
The type of the given transaction, such as "CT", "ET", "Nested CT", or "Nested ET".

TID tid — The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the TransactionType primitive fails and returns "NULL-TRAN-TYPE".

4.3.6.6 IsCommitted

The *IsCommitted* primitive checks whether the given transaction is already committed or not.

---

val = **IsCommitted**(tid)

BOOLEAN val — TRUE if the specified transaction was committed, otherwise FALSE.

TID tid	The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the *isCommitted* primitive fails and returns "FALSE".

4.3.6.7 IsAborted

The *IsAborted* primitive checks whether the given transaction is already aborted or not.

val = **IsAborted**(tid)

BOOLEAN val	TRUE if the specified transaction was aborted; otherwise FALSE.
TID tid	The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the IsAborted primitive fails and returns "FALSE".

4.3.7 File Management

Viewing a file as a set of long term persistent data, it is clear that an arobject can also fulfill this role. To create a file, an instance of the appropriate arobject can be created, and the filename can be bound to it. To erase the file, the *Kill* primitive will serve. Reading from and writing to the file can be performed using the appropriate operations of the arobject itself. Similarly, control functions (e.g. backspace, random placement, search) become operations of the arobject. The data to be stored in the file is merely contained in one of the arobject's *private data objects*. If this private data object is declared to be atomic, then the file will be treated as an *atomic* file and all of the file accesses must be performed from within a transaction scope.

4.3.7.1 File Access Interface

In order to avoid the low level (bare) access to a file arobject (i.e., an explicit invocation of an operation on a file arobject), ArchOS provides the following set of primitives which support conventional file access and control functions.

The *OpenFile* primitive opens a specified file with the given access mode, and the *CloseFile* file primitive closes the file. The *ReadFile* and *WriteFile* primitives provide a simple "byte stream" oriented read and write access to a file, respectively. The *CreateFile* primitive creates a file of a given

file type and the *DeleteFile* primitive deletes a file. The *SeekFile* primitive moves the current reading or writing position to a specified by bytelocation in the file.

---

fd = **OpenFile**(filename, mode)
val = **CloseFile**(fd)
nr = **ReadFile**(fd, buf, nbytes)
nw = **WriteFile**(fd, buf, nbytes)
val = **CreateFile**(filename, filetype)
val = **DeleteFile**(filename)
pos = **SeekFile**(fd, offset, origin)

FILEDESCRIPTOR *fd
A pointer to the file descriptor.

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

INT nr The actual number of bytes which were read.

INT nw The actual number of bytes which were written.

FILENAME filename
The name of the file.

ACCESSMODE mode
The mode for accessing the file. (e.g., "READ", "WRITE", "APPEND", "READLOCK", "WRITELOCK", etc).

FILE-TYPE filetype The type of the specified file such as "ATOMIC", "PERMANENT" or "NORMAL".

BUFFER *buf The buffer address.

INT nbytes The number of bytes to be read or written during a Read or Write operation, respectively.

LONGINT pos The current pointer's position in the file in bytes. If the seek action is done successfully, the value of pos must be a positive integer; otherwise -1

FILEOFFSET offset The offset value from the given origin point, in bytes.

FILEORIGIN origin The origin indicates the origin of the seek operation. (At the beginning, the current position, or the end of the file).

On Error: If the file does not exist or the file mode is not supported, then the *OpenFile* primitive fails and returns "NULL-FILE-DESCRIPTOR". If the file descriptor is not an opened descriptor, then a

close or read/write action fails and "-1" will be returned and a detailed error condition will be also set in the caller's error block. If a create/delete action fails, then "FALSE" will be returned and a detailed error condition will be also set in the caller's error block. If the specified file does not exist or the value of offset or origin is not in the proper range, "-1" will be returned and a detailed error condition will be also set in the caller's error block.

4.3.8 I/O Device Management

A normal I/O device access protocol should be similar to the file access protocol described in Section 4.3.7. To read, write or send a special command, the target device must be opened. Once all actions are done, it must to be closed. All device dependent commands can be sent to devices by using the *SetIOControl* primitive.

At the lowest level, I/O control and data transfer will be handled according to the hardware interface definition (e.g., memory read and write to memory mapped devices). The *IOWait* primitive can be used to wait for an interrupt from a particular device; at most one process may wait for a particular device interrupt at one time.

4.3.8.1 Basic I/O Device Access Interface

Typically, access to a device is performed by the following primitives. If a device is not readable or writeable, then a special *SetIOControl* primitive (see Section 4.3.8.3) must be used.

---

dd = **OpenDevice**(device-name, mode)
val = **CloseDevice**(dd)
nr = **ReadDevice**(dd, buf, nbytes)
nw = **WriteDevice**(dd, buf, nbytes)

DEVDESCRIPTOR *dd
A pointer to the device descriptor.

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

INT nr The actual number of bytes which were read.

INT nw The actual number of bytes which were written.

DEVNAME device-name
The device name.

ACCESSMODE mode
The access mode of the specified device such as "READ", "WRITE", etc.

BUFFER *buf The buffer address.

INT nbytes The number of bytes to be read or written during a *ReadDevice* or *WriteDevice* operation, respectively.

On Error: If the device does not exist or the device mode is not supported, the *OpenDevice* primitive fails and returns "NULL-DEV-DESCRIPTOR". If the device descriptor is not an opened descriptor, then a close or read/write action fails and "-1" will be returned, a detailed error condition will be set in the caller's error block.

4.3.8.2 IOWait

The *IOWait* primitive blocks the requestor process until the specified device completes an I/O action.

---

event-cnt = **IOWait**(dd, timeout)

INT event-cnt An event counter which indicates the number of basic I/O actions performed. If a negative number is returned, it indicates an error state.

DEV-DESCRIPTOR dd
The device descriptor.

TIME timeout The timeout value indicates the maximum execution time of this I/O function.

On Error: If the device descriptor is not an opened descriptor, a wait action fails, "-1" will be returned, a detailed error condition will be set in the caller's error block.

On Timeout: If an IOWait primitive cannot complete within the specified timeout value, a negative value will be returned.

4.3.8.3 SetIOControl

The *SetIOControl* primitive sends device control information to the specified device. It also receives status information from the device.

---

val = **SetIOControl**(dev-descriptor, io-command, dev-buf, timeout)

BOOLEAN val TRUE if this SetIOControl succeeded; otherwise FALSE.

DEV-DESCRIPTOR *dev-descriptor
A pointer to the device descriptor.

IO-COMMAND io-command
The io-command indicates a device-specific control command.

DEV-BUF *dev-buf The dev-buf indicates a pointer to a buffer which will be filled with device status.

TIME timeout The timeout value indicates the maximum execution time of this I/O function.

On Error: If the device descriptor is not an opened descriptor, then a control action fails and "FALSE" will be returned and a detailed error condition will be also set in the caller's error block.

On Timeout: If an *SetIOControl* primitive cannot complete within the specified timeout value, a negative value will be returned.

4.3.9 Time Management

In ArchOS, the time management not only provides a basic access to the system real time clock which is maintained in non-volatile storage, but also supports primitives which provide the basic functions of the time-driven (process) scheduling.

The *GetRealTime* primitive obtains the system's current real time clock value, while the *GetTimeDate* primitive fetches the current absolute time and date. The *Delay* primitive delays the caller's execution for the specified time period. The *Alarm* primitive also postpones the caller until the specified time of day and date. The *Delay* and *Alarm* primitives provide aperiodic time dependent processing control, while periodic repetitive processing will normally be controlled using the Policy primitives (see Section 4.3.10).

4.3.9.1 GetRealTime

The *GetRealTime* primitive returns the current real time clock value in microseconds.

---

rtc = GetRealTime()

REALTIME rtc Value of current real-time clock in microseconds. This value may be used to compute time values for use in delay or alarm primitives.

4.3.9.2 GetTimeDate

The *GetTimeDate* primitive returns the current absolute time and date. The time value accuracy will be limited by delays in the calibration entry performed by the application, and cannot be expected to return exactly the same value simultaneously at every node. ArchOS will maintain this information on a best effort basis.

(time, date) = **GetTimeDate()**

TIME time — Value of current time of day in microseconds from midnight.

DATE date — Julian date of this day.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.3 Delay

The *Delay* primitive delays a specified length of time (in microseconds), then returns the current date and time when the delay has been completed and execution has resumed. This primitive can also be used to specify a deadline, by which time a deadline milestone must be reached, as well as an estimate of the amount of processing time that will be required to reach the deadline milestone. (ArchOS may also make estimates about this processing time based on observations of earlier executions.) Finally, the client may use the *Delay* primitive to mark the arrival of the process at the deadline milestone referred to as the current deadline (while optionally defining the next deadline milestone).

(time, date) = **Delay**(delaytime, deadline, util, dlflag)

REALTIME delaytime
: Time to be delayed starting at the present time, in microseconds. No delay if this value is not positive; in this case it replies immediately.

REALTIME deadline
: Elapsed time in microseconds from delaytime by which the deadline milestone must be reached. If this value is not positive, the deadline remains unchanged.

REALTIME util
: Estimated execution time of this process in microseconds which will be required before the deadline is reached. If this value is 0 or less, the current processing time estimate remains unchanged. The use of this value by ArchOS is defined by the policy mechanism (see Section 4.2.6).

TIME time — Current time of day in microseconds since midnight.

DATE date — Julian date of this day.

BOOLEAN dlflag — Client sets to "TRUE" if this call is intended to mark the arrival of this process at the milestone referred to as the current deadline.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.4 Alarm

The *Alarm* primitive waits until the specified time of day and date, then replies with the current time and date. If the specified time has already passed, it replies immediately. (The accuracy of the day and date is as described in the *GetTimeDate* primitive above.) Like the *Delay* primitive, the *Alarm* primitive can also be used to specify a deadline, by which time a deadline milestone must be reached, as well as an estimate of the amount of processing time required to reach the deadline milestone. (ArchOS may also make estimates about this processing time based on observations of earlier executions.) Finally, the client may use the *Alarm* primitive to mark the arrival of the process at the deadline milestone referred to as *the current deadline* (*while optionally defining the next deadline* milestone).

(time, date) = **Alarm**(alarmtime, deadline, util, dlflag)

REALTIME alarmtime — Time and Date at which processing of this process is requested to resume.

REALTIME deadline — Elapsed time in microseconds from delaytime by which the deadline milestone must be reached. If this value is not positive, the deadline remains unchanged.

REALTIME util — Estimated execution time of this process in microseconds which will be required before the deadline is reached. If this value is 0, the current processing time estimate remains unchanged. The use of this value by ArchOS is defined by the policy mechanism (see Section 4.2.6).

TIME time — Current time of day in microseconds since midnight.

DATE date — Julian date of this day.

BOOLEAN dlflag — Client sets to "TRUE" if this call is intended to mark the arrival of this process at the milestone referred to as the current deadline.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.5 SetTimeDate

The SetTimeDate primitive sets the current absolute time and date. The date will be written into the ArchOS data base and continually adjusted by ArchOS. ArchOS will maintain this information on a best effort basis. This primitive should be used only once, when the application is initiated.

---

val = **SetTimeDate**(time, date)

BOOLEAN val	TRUE if time and date were set; otherwise FALSE.
TIME time	Value of current time of day in microseconds from midnight.
DATE date	Julian date of this day.

On Error: If the time and date were already set by this primitive, they remain unchanged, and val is set to "FALSE".

4.3.10 Policy Management

In ArchOS, a policy management is carried out by a *policy set arobject* and a set of *policy modules*. The policy set arobject must exist in a distributed program and maintain a directory of the policy modules. The actual policy will be implemented as a separate policy module in ArchOS. A set of *policy attributes* are also maintained by the policy set arobject and ArchOS and will be to referred by the policy modules.

A *SetPolicy* primitive specifies a new policy module to carry out a designated policy in the applications's policy set arobject. If no *SetPolicy* primitive is invoked, ArchOS provides a default policy module for the application program. A *SetAttribute* primitive sets an attribute into a new value.

---

val = **SetPolicy**(policy-name, policy-module)

val = **SetAttribute**(attr-name, attr-value)

BOOLEAN val	TRUE if the specified policy was set properly; otherwise FALSE.

POLICY-NAME policy-name
The name of the policy to be set. (Well-known name to ArchOS, such as *SCHEDULE*.)

POLICY-MODULE policy-module
The name of the policy module which contains the policy.

ATTRIBUTE-NAME attr-name
The name of attribute to be set.

ATTRIBUTE-VALUE attr-value
The actual value for the attribute.

On Error: If there is no policy name or policy body, the SetPolicy primitive fails and a "FALSE" will be returned. The SetAttribute primitive fails if a specified attribute is not defined or inappropriate values is assigned.

4.3.11 System Monitoring and Debugging Support

ArchOS provides system monitoring and debugging primitives in order to control the complexity of application development in a distributed environment. A system monitoring facility can provide for tracking the behavior of arbitrary arobjects or processes in the system. The communication activity among arobjects can be also monitored by intercepting the selective message.

4.3.11.1 Freeze and Unfreeze

A *FreezeAllApplications* primitive stops the entire activities of caller's application, and a *FreezeNode* primitive halts all of the cleint's activities in a specific node. To resume client's application, *UnfreezeAllApplications* or *UnfreezeNode* will be used.

A *FreezeArobject* primitive stops the execution of an arobject (i.e., all of its processes), and a *FreeseProcess* primitives halts a specific process for inspection. An *UnfreezeArobject* and *UnfreezeProcess* primitive resumes a suspended arobject and process respectively. While a process is in a *frozen* state, many of the factors used for making scheduling decisions can be selectively ignored. For instance, a timeout value will be ignored by specifying a proper flag in the *Freeze* primitive.

---

val = **FreezeAllApplications**()
val = **UnFreezeAllApplications**()
val = **FreezeNode**(node-id)
val = **UnfreezeNode**(node-id)
val = **FreezeArobject**(arobj-id [, options])
val = **UnfreezeArobject**(arobj-id [, options])
val = **FreezeProcess**(pid [, options])
val = **UnfreezeProcess**(pid [, options])

BOOLEAN val — TRUE if the primitive was executed properly; otherwise FALSE.

NODE ID node-id — The node id indicates the actual node which wil be stopped.

AID arobj-id — The unique arobject id of an arobject instance.

PID pid — The process id of the target process.

FREEZE-OPT options
The options indicate various selectable flags such as a timeout freeze/unfreeze flag.

On Error: An error condition, such as non-existent node id, arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the target node, arobject, or process is already unfrozen, then the *Unfreeze* action will be ignored and FALSE will be returned along with the appropriate error information is in the error block.

4.3.11.2 Fetch and Store Arobject and Process' Status

A *Fetch* primitive inspects the status of a running or frozen arobject or process in terms of a set of frozen values of private data objects. The specific state of the arobject or process will be selected by a data object id. The state includes not only the status of private variables, but also includes process control information.

---

fval = **FetchArobjectStatus**(arobj-id, dataobj-id, buffer, size)
sval = **StoreArobjectStatus**(arobj-id, dataobj-id, buffer, size)
fval = **FetchProcessStatus**(pid, dataobj-id, buffer, size)
sval = **StoreProcessStatus**(pid, dataobj-id, buffer, size)

INT fval — The actual number of bytes which were fetched.

INT sval — The actual number of bytes which were stored.

AID arobj-id — The unique id of the arobject instance.

PID pid The process id of the target process.

DATAOBJ-ID dataobj-id The dataobj-id indicates the private object or system control status of the target arobject/process.

BUFFER *buffer A pointer to the buffer area for storing the returned data object value.

INT size The size indicates the buffer size in bytes.

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the fetched data object is larger than the specified buffer size, then the content will be truncated. For the storing operations, the data object size must be equal, otherwise the value will not be replaced.

4.3.11.3 Kill Arbitrary Arobject/Process

A *GlobalKill* primitive can destroy an arbitrary arobject or process in the system.

---

nproc = **GlobalKillArobject**(arobj-id [, options])
val = **GlobalKillProcess**(pid)

INT nproc The actual number of killed processes.

BOOLEAN val TRUE if the specified process was killed; otherwise False.

AID arobj-id The unique id of the arobject instance.

PID pid The process id of the target process.

GKILL-OP options The options indicate various control options. For example it can indicate whether the caller stops every time after killing a single process or not.

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the target arobject or process was already killed, then no action will be performed and "FALSE" will be returned.

4.3.11.4 Monitor Message Communication Activities for Arobject/Process

The *CaptureComm* primitives capture on-going communication messages from the specified arobject or process. A *CaptureCommArobject* primitive captures all of the incoming request and outgoing reply messages to a specified arobject and can select a target message based on the name of the operation. A *CaptureCommProcess* primitive captures all of the incoming messages and outgoing reply messages for

The *WatchComm* primitives are similar to CaptureComm primitives except that all of the monitored messages are duplicated, not captured.

---

val = **CaptureCommArobject**(arobj-id, commtype, requestor, opr)
val = **CaptureCommProcess**(pid, opr)
val = **WatchCommArobject**(arobj-id, commtype, requestor, opr)
val = **WatchCommProcess**(pid, opr)

BOOLEAN val — TRUE if the monitoring action was initiated successfully; otherwise FALSE.

AID arobj-id — The unique id of the arobject instance.

PID pid — The process id of the target process.

MSG-Q commtype — This indicates either "REQUEST" or "REPLY" type.

AID requestor — The aid of the communicating arobject.

OPE-SELECTOR opr — The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block.

4.3.11.5 SetErrorBlock

A *SetErrorBlock* primitive sets an error block in a process's address space. A user error block consists of a head pointer and a circular queue. The head pointer contains a pointer to an entry which contains the latest error information in the circular queue. After the execution of this primitive, a client can access the detailed error information from the specified error block.

It should be noted that the *SetErrorBlock* primitive will be executed at the process creation time (in the library routine), so that the system default error block will be set automatically.

---

val = **SetErrorStack**(errblock, blocksize)

BOOLEAN val TRUE if the error block is set successfully; otherwise FALSE.

ERROR-BLOCK *errblock
The address of error block.

INT blocksize The size of error block in bytes.

On Error: If the address of the error block is not valid, then the primitive fails and returns FALSE.

4.4 Rationale for the ArchOS Client Interface

This chapter is organized in parallel with the organization of the first chapters of this document. A rational approach to reading this chapter would be to remove this chapter from the document placing it side by side with the remaining sections of the document and then reading it in parallel with the points made in the remaining sections.

4.4.1 Introduction

The preceding sections describe in some detail the specifications for the ArchOS client interface. Most specification documents would end here having as completely as possible specified the operations and the expected responses of the operating system. This specification, and others like it, however, embody the results of a large number of decisions. This chapter is designed to describe the rationale for the decisions made. Obviously not every decision can be completely described here. It's entirely possible that there will be a number of important decisions which we will not describe, but our attempt is to describe all those trade-offs that we have consciously made with respect to the overall functions involved. We will try to identify alternative configurations that we had discussed and will try to identify the reasons for the particular decisions made.

4.4.2 ArchOS Computational Model

The rationale for this section must perhaps be the most incomplete, since there are of course, an large number of choices one can make for the computational model of a distributed system. Distributed systems have been built using models varying from systems built on a star configuration where one node is completely in charge of the system and the others operate in a slave relationship to autonomous systems; networks of multiple processors tied together and communicating to solve

either a common problem or a large set of disjoint problems. The purpose of ArchOS is to build, as we have stated, a distributed computer (i.e., a set of processing nodes which, operating together, act as a single functional entity to solve a particular application problem) Thus we needed a computational model that would reflect the unity of purpose inherent in such a concept.

In addition, we were concerned with the software engineering aspects of the application design. In many existing real time systems (uniprocessors as well as multiple processor systems) we find that there is a strong tendency to design the application software along the lines of the operating system interface, thus causing application partitioning to occur in the program at points that do not correspond to the application problem itself. This effect is perhaps most clearly illustrated with a standard Navy real time operating system in which each event must be handled by a user process specifically designed to handle the event. Thus a single process may not handle both time and I/O events, and sequential I/O operations must be performed by separate process invocations, resulting in a very disjoint (and non-modular) program structure. This creates a problem with the reliability and maintainability of the application system. Although ArchOS is not a production system, we expect that eventually a production system will be built along the lines explored by this Archons research and therefore we would like to start with a computational model which will lend itself to good software engineering practices. In today's technology we felt this included first of all the need to define the application as instantiations of abstract data types, but here we wanted to ensure that the abstract data types we produced could exploit our distributed environment.

It is also true that in existing real time systems, and particularly in command and control systems such as we are considering for our target applications, the programs are frequently very large and are constructed by large teams of programmers. We would like to have a computational model which would allow not only good software engineering practices with respect to small programs (i.e., programming in the small), but also with respect to the problems of building software in the large. Hence, we have chosen a large primary entity for our computational model, the arobject. The size of the arobject and its clean interface lends itself to a reasonable organization of software engineers for development purposes. In addition, the arobject to arobject interface from is sufficiently simple to allow a software engineering organizational break-out along arobject lines. This is intended to simplify not only the software development of a highly modular application, but also the test and evaluation phase of such a large system.

4.4.2.1 Principal Components

Clearly, our choice for the top level principal component (i.e., the distributed program), is a natural one in light of the fact that the Archons project is producing a distributed computer and an operating system for that computer. We see the distributed program as being a single entity with respect to its overall function, although it is made up of a number of much smaller components. We have chosen not to be concerned with the execution of more than one distributed program, although we have not prohibited it since more than one distributed program may need to be present during application testing. It may be argued that we are merely playing with words here, and in a sense we are, but the only distinction between a single distributed program and more than one is that resource allocation priorities between arobjects in the distributed program are defined, but resource allocation priorities between arobjects in different distributed programs cannot be determined. So it is possible to take two programs which are disjoint and merge them together by defining these interrelationships. Clearly, although we do plan to be able to operate ArchOS in an environment with more that one program running, we cannot claim that resource management decisions will be made fairly between the two systems.

As we have stated, the arobject is a distributed abstract data type. We have taken from the Ada model the concept of a separated specification and body. Similarly, the specification section is the only portion which is visible in the computational sense from one arobject to another. We see the arobject as a distributed abstract data type which can be instantiated more than once in the distributed system.

We should note that an instance need not be resident on a single node. We felt that tying the instance of an arobject to a given node would render inflexible the potential use of an arobject to handle a distributed abstract data type. The decision to limit objects to a single node has been made in the design of a number of systems, such as the Eden system [Almes 83], in which an Eject (conceptually somewhat similar to an arobject) must be entirely resident on a single node. Obviously, the decision to be resident on a single node has the advantage that a common address space can be used for the entire object. We felt that not requiring the entire arobject address space be contained on a single node would give us great flexibility in application design, so we deliberately chose not to require such an organization.

By allowing more than one instance of a given arobject, thus to be bound to a single reference name, we have made it possible for a higher level arobject to completely handle a distributed abstract data type. One could envision, for example, a partially replicated directory existing on a number of nodes or possibly the entire set of nodes in a distributed system, but being handled by a single

arobject distributed over that network. The specification portion identifies the entire interface to such a group of arobjects by external arobjects, but processes within the arobject itself can communicate with each other across the various nodes regardless of the node boundaries. Obviously, the performance of such a system must be taken into account and the arobject will have options at its instantiation time with respect to ensuring that particular subcomponents (processes, private abstract data type instances, and so on) are resident on common nodes or separate nodes as dictated by its requirements.

The ArchOS file system illustrates some of the arobject lifetime CHARACTERISTICS. We have chosen at this point to make very simple our concept of the file system by simply using permanent instances of file arobjects [Almes 83]. Such arobjects can be instantiated or killed as required and the set of their permanent arobject id's then becomes, in effect, a directory of the files being kept on the system. The operations of these arobjects would provide the primitives required to access, modify, update and delete the data within the arobject. We would envision, however, that not all arobjects would be permanently instantiated. In fact, quite a number of them might be instantiated during the system start-up operation and would die automatically when the program is terminated for any reason, such as by powering off the system. Thus, an arobject instance might never have a permanent form in existence.

The arobject body is intended to implement the operations described in the arobject specification. It should be noted that the private abstract data types comprise the only form of shared data between processes within an arobject. These are classical abstract data types which contain the data itself and which can be manipulated only via the defined procedures in the abstract data type. Encapsulating atomic or permanent data within a private abstract data type allows controlled access to shared data. ArchOS will require that any procedures which access atomic data must have a transaction open at the time of access. Thus, ArchOS can control the sharing of this data, and when the transaction is committed or aborted, ArchOS will ensure that the atomic data is correctly forced to appropriate stable storage. We also note that because of our arobject distribution provisions, the private data can be partitioned among multiple nodes, although each individual private abstract data type instantiation must be fully resident on a single node. In this way remote procedure call semantics, including parameter passing by value, will be used if the processes not co-located with the private procedure on the same node. It is intended that defining the private abstract data types in this way will allow us to directly implement such distributed applications as a partially replicated directory, and in fact, an example of such an implementation can be found in Appendix A of this document.

At this point, let us observe that there is no required correlation between the operations of an

arobject and the processes defined in an arobject body. It's anticipated that one or more processes in each arobject body will run continuously, pausing from time to time to check for the existence of an operation request from another arobject. If such a request is not found, that process could block awaiting another such request. Thus, the operation can be handled by any process within the arobject, which allows us to have a set of processes with essentially identical function, handling operations in any manner that the application desires. The binding between these operations, then, is late and is handled dynamically by the appropriate processes. If, of course, the designer wants to couple the processes to the operations he need only specify his *Accept* parameters to identify which operations he wishes to accept at each point and therefore he can bind them as early as he wishes. A cost of this approach is that the potential implementation error of omitting the handling of some operation in an arobject cannot be detected at system generation time, but we feel that the benefit in terms of modularity and concurrency greatly outweighs this problem.

Processes within an arobject, of course, may communicate by using the *CreateProcess* primitive to create processes and passing parameters at that time, or they may invoke operations within the arobject using a *Request* primitive, using either operations from the specification part, or private operations from the body. These private operations are designed to make it possible to place operation requests into the incoming queue from internal processing which are separable from those placed by external arobjects. Obviously processes within an arobject can also communicate via shared data, using the private abstract data types. Any scheduling or mutual exclusion which must be handled with respect to this shared data would be handled within the abstract data types.

In addition, we have allowed for inclusion of private arobjects to be defined within the body of an arobject. Such a private arobject will not be visible to external arobjects, and could be used for partitioning operations within an abstract data type. This technique could be used for an implementation of a partially replicated directory by placing each partition in its own arobject and using the outer arobject to distribute the lookup and scheduling for the arobjects containing the data. (See Appendix A Solution 2.)

One may, of course, question our contention that these processes are lightweight as we have defined them; that their scheduling overhead will be small relative to the speed of the machine. This is certainly our intention, but we will have to weigh this against the implementation requirements to obtain the interfaces we need. It is our intention, however, to minimize the state required to be constructed in the scheduling of a process. In addition, we envision eventually building special purpose hardware for hosting ArchOS, one objective of which will be to optimize the creation of processes and the resulting context swap overhead.

4.4.2.2 Communication Facilities

4.4.2.2.1 Rationale for Accept/Request Rendezvous Mechanism

The *Request/Accept/Reply* primitives were selected to provide the means of communication among arobjects. These primitives allow communicating arobjects to rendezvous, rather than providing a master/slave relationship for all communications.

The master/slave paradigm was rejected because it seemed too restrictive. Consider that a major goal of ArchOS is to support decentralized resource management by collections of resource managers, which negotiate to reach a consensus regarding a particular management decision. The communications involved in carrying out negotiations among resource managers are not master/slave in nature; they are better characterized as communications among peers.

The *Request/Accept* rendezvous captured this sense of peer communication--in order for communication to take place, both parties involved must explicitly act; the requestor cannot force a process to perform an *Accept* for a particular invocation.

Both blocking and non-blocking request primitives are provided to give the client a very flexible communication facility.

4.4.2.2.2 Rationale for Broadcast Request Capability (*RequestAll* Primitive)

The *RequestAll* primitive provides the capability to broadcast a request to multiple arobjects. This feature of the inter-arobject communication facility was very directly shaped by the anticipated structure of ArchOS' internal distributed data entities.

It is assumed that the ArchOS operating system will make use of various data objects that must be highly reliable and available. Such objects can be constructed by means of data replication and distribution throughout the operating system, with appropriate use of the ArchOS transaction facilities. In addition, ArchOS will almost certainly contain multiple instances of various servers (for instance, file servers and name servers). If data is either partially or completely replicated at multiple locations or if replicated servers are available at multiple locations, it seems natural to use broadcasts as a common mode of communication involving these replicated entities. For example, a new entry in a name table might be broadcast to all of the arobjects that contain a portion of that name table. Each partially redundant table fragment could then be updated appropriately.

4.4.2.2.3 Rationale for Intra-Arobject Request Capability

Despite the number of communication primitives provided by ArchOS, we wanted to keep the model of communication among entities as uniform as possible. As a consequence, ArchOS does not support a direct process-to-process communication capability. All processes communicate only with arobjects, by means of the Request/Accept mechanism, even when a process wishes to communicate with another process in the same arobject. (This restriction is a consequence of the fact that we do not want the user of an arobject to know about the implementation of another arobject (including the number of processes and the process id's in that other arobject). The arobject operation invoker can only see the specification, not the implementation, of the arobject to be invoked. Therefore, process-to-process communication between two distinct arobject instances is not possible. And in the interests of uniformity and simplicity, we also decided not to allow such communications to take place within a single arobject instance.)

In order to allow processes in a single arobject to perform additional operations, beyond than those that are visible to other arobjects, it was necessary to provide an additional set of operations that can only be invoked by the processes in that arobject. These operations are called the *private operations* of the arobject, and they are invoked in exactly the same manner as operations on other arobjects.

4.4.2.2.4 Rationale for Invocation Parameter Passing

ArchOS *Request* parameters are passed to the receiving arobject using call-by-value semantics. This is the only reasonable way to pass parameters since arobjects do not share any address space.

The only exception to this rule comes in the case of an invocation by a process of a private operation. In that case, the processes can have intersecting address spaces due to the presence of shared private data (in private abstract data type instances) in the arobject. As a result, it is possible for such *Request* and *Reply* messages to contain references to common objects (for example, the names of private data type instances).

4.4.2.3 System Load and Initialization

The tradeoffs involved in this section are fairly simple. We expect that the system would be loaded in a conventional manner from external storage (e.g. disk) associated with some of the nodes in the system, and we expect that nodes not containing local external storage would obtain load data from neighboring nodes. We expect this to be an internal ArchOS design decision, which will therefore be specified at a later time.

The question to be answered in this section is how the application program would be initialized once ArchOS is fully initialized and has determined its own status. We have taken the position at this

point that an application program will do this by the definition of a unique application arobject (the *Root* arobject), which ArchOS will expect to find on one or more nodes. This arobject will then determine its own status, including finding out if other copies of itself exist (or insuring its own uniqueness if needed). This arobject will then *Create* the other arobjects needed to bring up the application program.

This is a simple approach which has been planned to maximize application program flexibility at initialization time, and to eliminate the need for operator intervention other than that required by the application itself.

4.4.2.4 Transactions

ArchOS is a highly decentralized, real time operating system designed to support highly decentralized, real time applications. In fact, the operating system itself can be viewed as a highly decentralized, real time application built on top of the kernel support facilities. We have attempted to view ArchOS in this way at various times, and that has led us to view the arobject as a computational entity that we would use *within* ArchOS (insofar as possible), as well as at the application level. While specifying the services to be provided by ArchOS and its behavior under all conditions, it became apparent that ArchOS' internal system data objects should possess several characteristics. In particular, the following characteristics were desired:

- Often, several different arobjects will operate on specific data items (directories, queues, and so on). These shared data objects will reside in an arobject, so access can be coordinated by the normal arobject communication facilities (particularly *Accept* primitives), as well as by means of customized code in the arobject's processes. But, as the following points will illustrate, a higher level coordination will often be desirable.

- At times, it is necessary to change several different, yet related, data items as a unit (for example, updating all of the copies of an entry in a partially replicated directory or moving an element from one queue to another). If such atomic updates could be performed, then it would be much easier to transform one consistent state of a set of data items to another consistent state.

- Some data items must be permanent; that is, their state should be reliably maintained for the life of the system.

These attributes could all be provided by ArchOS by means of the communication facilities in conjunction with custom-written code and appropriately defined locks or semaphores and critical regions. However, we desired a more structured approach. All of the above capabilities are supported by traditional database transaction systems [Gray 77]. In fact, such transaction systems can provide even more powerful properties (specifically, failure atomicity and/or serializability). It was felt that this additional structure would ease the programmer's burden, while also decreasing the

chances for programming errors, by making modular programming more natural. (Indeed, the use of compound transactions promotes modular construction of programs by causing the transaction author to think in terms of consistency preserving transformations on sets of atomic data objects, thereby causing the program's data to be partitioned into a number of modular atomic data sets. Also, transaction systems in general can aid the programmer in another important way: the processing carried out in order to commit or abort a transaction can handle a great deal of lock-related bookkeeping, even though the programmer must explicitly obtain locks on all of the atomic data items. This frees the programmer to consider the correct behavior of the transactions being written, without giving unnecessary consideration to interactions with other transactions in the system. However, this is not to say that the programmer does not have to be concerned at all with locks and locking protocols; rather, it is intended to point out one aspect of lock management that the programmer does not have to handle. Using the current ArchOS locking protocols, there are still a number of decisions concerning locks that the programmer must make--for instance, whether to use a discrete locking protocol or a tree locking protocol, how to organize the locks in a lock tree, what data items are associated with a given lock, and so on. Section 4.4.2.7 deals with some of these issues in more detail.)

The above discussion explains why a transaction facility was included for use *within* ArchOS. Since the ArchOS clients are also interested in producing highly decentralized, real time programs, it was felt that it was appropriate to extend these primitives to the clients as well.

Of course, there are arguments against using a transaction facility within an operating system. One major objection is that system performance could be greatly reduced (as compared to a system that does not use a low-level transaction facility). This is due to the fact that occasionally the system will have to suspend the processing of a specific transaction while data is being copied from main memory to a (virtually) permanent medium; there will also be overhead associated with the initiation and conclusion of each transaction. (The impact of mutual exclusion on system performance is not mentioned in the preceding discussion since some form of mutual exclusion must be present in any system containing shared data objects, whether it includes transactions or not.)

It seems inevitable that a performance penalty will be incurred by the use of transactions, but it is hoped that the gains in the area of data permanence, system consistency, high availability, and reliability will be worth the price. However, three other decisions were made to address the problem of performance losses due to the use of transactions in ArchOS:

- not all processing must take place within transactions.

- only specifically designated data items would be defined as *atomic* -that is, only those items would be permanent, failure atomic, and so forth. (This decision forces the arobject writer to explicitly indicate which data items must be atomic and has a great influence on the types of steps that appear within a transaction. For instance, it would be a questionable, if not wrong, programming practice to use non-atomic variables to pass values from one nested transaction to another. An example illustrating this point is shown in Section 4.4.3.5.)

- a new type of transaction, the compound transaction, is included to increase the potential degree of system concurrency.

4.4.2.5 Rationale for the Inclusion of Compound Transactions

Compound transactions addressed two great concerns regarding the use of transactions, both within ArchOS and by ArchOS clients. One of these concerns, performance, has already been mentioned. Since ArchOS allows transactions to be nested arbitrarily (interleaving elementary and compound transactions as desired), the use of some compound transactions can increase system concurrency. This is due to the fact that compound transactions release all of the locks that they, or any of their child transactions, have set at the completion of the execution of the compound transaction. Thus, the resources that are controlled by these locks are often free to be used by other processes prior to the completion of all of the processing associated with a given transaction.

The second concern addressed by compound transactions in ArchOS is that of system integrity and liveliness. In traditional database systems which support nested transactions, all of the locks obtained by child (nested) transactions are passed to their parent transactions and kept until the completion of the highest level transaction (at which time the transaction is either aborted or committed). This approach was not suitable for ArchOS, where most of the operating system primitives are actually expected to be implemented using transactions. If a client transaction contained operating system primitive calls which were implemented using traditional nested transactions, then on the completion of the primitive call, the client transaction would receive any locks that the system primitive transaction(s) had obtained for system resources. The client could subsequently attempt to manipulate the system resource or could simply hold the lock on the system resource for an arbitrarily long time. Both of these possibilities were disturbing; but, both were also preventable by proper use of compound transactions. If each ArchOS primitive is not just a transaction, but rather a compound transaction, then no locks on system resources will ever be returned to the client transaction. In this way, compound transactions are used in ArchOS to build a "firewall" between the operating system and the client.

Of course, compound transactions have some disadvantages associated with them as well. For instance, it is not possible to simply change an arbitrary elementary transaction to a compound

transaction without consideration of recovery issues. (Elementary transactions in ArchOS correspond to traditional nested transactions in database systems.) During the course of its execution, each compound transaction may invoke a number of arobject operations and/or operating system primitives. At the completion of the execution of the compound transaction, all of the locks that have been obtained during transaction processing are released (despite the fact that the compound transaction may be nested within another transaction). In the event that a higher-level transaction aborts after the compound transaction has committed, it is not possible to guarantee that all of the operations performed by the compound transaction can be properly "undone" (in the sense of traditional nested transactions). For instance, it is possible that another transaction has read, and acted upon, data that represented the outcome of the compound transaction after it had committed (and thereby released all of its locks), but prior to the execution of its compensation action. Such a situation could never arise in a traditional transaction system, but it certainly could happen in the ArchOS transaction system.

The ArchOS compensation action for a committed compound transaction consists, in part, of the execution of a set of compensation operations associated with the arobject operation invocations and operating system primitive invocations made during the course of execution of the compound transaction. While these compensation operations may attempt to approximate the effects of the traditional transaction "undo" operations, they cannot guarantee that the compensation will result in the same system state as would have resulted if only nested elementary transactions been used. Rather, the system state is transformed to a state that is equivalent to the state that would have resulted if all of the other concurrent transactions in the system had been processed in the absence of the aborted compound transaction. (See Section 4.2.4.1 for additional discussion of this point.) If such compensation operations can be constructed and the weaker guarantees concerning the system state in the case of the abortion of a higher-level transaction are acceptable to a transaction author, then compound transactions can be used for a given application; however, if these conditions are not sufficient, then elementary transactions must be used.

A few examples can be used to demonstrate cases in which compound transactions are or are not appropriate based on the ability of compensation actions to provide the required semantics.

First, consider a case in which compound transactions are appropriate: the dequeue operation of a weak queue. In such a queue, the first-in-first-out ordering of elements in a strong queue is weakened; it is acceptable to alter the order in which queue elements are removed from the queue by the dequeue operation. As a result, it is possible to dequeue elements by means of a dequeue operation based on a compound transaction. This operation simply returns the head element of the

queue and then releases any locks obtained in the process of dequeuing that element. The compensation operation associated with this dequeue operation is also quite simple: the element that was previously dequeued is returned to the head of the queue. Because of the semantics of the weak queue, these operations are acceptable. It is unimportant whether or not any other dequeue operations occurred during the interval between an element being dequeued and subsequently being requeued because a strict ordering of the queue elements is not required.

Second, consider a case in which a compound transaction is inappropriate: the enqueue operation of a weak queue. In this case, there is a visibility problem--that is, if compound transactions were used to implement the enqueue operation on a weak queue, it would be possible for other transactions to view the queue in states that represent partial results of computations that are subsequently aborted. As a result of viewing such states, it is possible that those transactions will alter the state in an inappropriate way. This situation is illustrated by the use of a compound transaction to implement the enqueue operation for a weak queue. Such an enqueue operation would take an element passed to it and append it to the tail of the queue, releasing any locks obtained at the completion of the operation. The most obvious compensation operation for this enqueue operation would locate the desired element in the queue and remove it, thereby attempting to make it appear as though it had never been there. However, this is not a sufficient compensation action since it is possible that an element may be enqueued and later dequeued before the compensation action is able to be executed. In such a situation, the only way to provide the required semantics for the weak queue is to abort the transaction that dequeued the element. Yet this presents the possibility of cascading aborts, and ArchOS cannot permit cascading aborts to occur. Due to this visibility problem, compound transactions are not appropriate for the implementation of the enqueue operation.

Another disadvantage associated with the use of the compound transaction is also related to the compensation mechanism: programmers must explicitly write the compensation operations corresponding to the arobject operations for a given arobject. The concept of this type of transaction is quite new, and we are not yet certain about the nature of the actions to be performed by a typical compensation routine. (In fact, there are many issues that we do not fully understand with respect to compound transactions: the number and variety of applications for compound transactions, the amount of work required to define appropriate compensation actions, the form such actions should take, the impact of compensation actions on the ability of ArchOS to make guarantees about real time behavior, the level of concurrency that can be achieved using compound transactions, and so on.) So at this point, we have decided that ArchOS will initially support only programmer-coded compensation actions. (This should be contrasted with the case of traditional nested transaction

database systems or, equivalently, nested ArchOS elementary transactions. In these cases, all of the "undo" or "redo" types of operations are determined and performed by the transaction facility for any user-written transaction.) However, ArchOS will provide support to automatically compose these basic compensation actions in order to facilitate the construction of arbitrary higher-level transactions (either compound or elementary transactions) that invoke arobject operations based on lower-level compound transactions.

4.4.2.6 Rationale for the Transaction Syntax

Two potential formats were considered for the syntactic definition of transactions: an in-line format (in which the transaction would be delimited in the body of the surrounding text by some keywords) and a procedural format (in which a transaction was defined as a separate entity--such as a function or procedure--and was "called" by the transaction initiator).

There was no overwhelming reason for choosing one format over the other. This issue seemed to be largely one of stylistic preference, not performance or functionality. Assuming that a typical transaction is relatively short, the in-line format has the advantage of showing the actual transaction steps to a reader of the code; on the other hand, the procedural format could be parameterized in the hope of avoiding the duplication of definitions that might occur with the in-line format. (This seems to parallel the arguments for using macros or subroutines in a given application.) Our preference was to use the in-line format since it appeared to be more readable and compact.

Once the determination to use an in-line format was made, we needed to pick a specific format. We felt strongly that transactions should not span multiple arobjects--that is, a transaction should not be initiated by one arobject and later completed by another arobject. Since the arobject is the basic unit of program construction, transactions that span arobjects hardly seem to promote modular program construction techniques. In fact, we felt that a transaction should begin and end in a single process. The reasoning for this decision is a simple extension of the argument previously given for requiring the transaction to begin and end in a single arobject. The *ET{...}* and *CT{...}* syntactic structures selected for use in ArchOS force a transaction to begin and end in a single process, while other possible structures (such as arbitrarily placed *BeginTransaction* and *EndTransaction* delimiters) did not.

4.4.2.7 Rationale for Lock Support Decisions

Several important decisions were made with respect to the support to be provided to the client in terms of obtaining locks on shared data objects. This portion of the rationale will deal with three of the most important decisions: (1) the decision to support both a discrete locking protocol and a tree

locking protocol [Silberschatz 80]; (2) the decision that the client must explicitly set locks ; and (3) the decision to allow the client to explicitly release locks within a transaction.

ArchOS supports both discrete locks and tree locks because we feel that each type of lock addresses a different set of client needs, and neither type alone addresses all of these needs. For instance, the tree lock is supported because it is able to make an important guarantee about the nature of the computations that use exclusively tree locks from a single lock tree: if a computation, C, obeys the tree lock accessing rules and if all of the other computations that obtain locks in that tree release them in a finite length of time, then computation C will also complete in finite time *without the occurance of deadlocks*. We believe that the guarantee that a computation will take place without the possibility of a deadlock is extremely important and justifies the support of tree locks in ArchOS.

However, tree locks cannot be the only locking mechanism in ArchOS. In defining the tree structure to be used in connection with tree locks, the client is explicitly specifying the legal access patterns for locks in the tree. This may be a straightforward process when the client is dealing with a small collection of related data items, but appears to be intractable when dealing with all of the locks in the system. (If it were desired to guarantee that the entire system would be deadlock-free, then it would be necessary to place all of the locks in the system in a single lock tree. Specifying a rational tree structure for such a large number of often loosely related items seems impossible.) This leads us to support discrete locks as well as tree locks. (Actually, ArchOS will provide support for multiple lock trees defined by the client, as necessary.)

In fact, we could build ArchOS without discrete locks, using a forest of tree locks. Yet, once it was decided that we could not include all of the locks in a single monolithic tree, it seemed to be worthwhile to allow the more traditional discrete lock to be used as well. Presenting the view to the client that each discrete lock is actually a degenerate (single node) tree lock seemed to be unnecessarily complicated. (Although that may be the manner in which discrete locks are actually implemented.)

The next major point to be discussed is the justification for the rule that a client must explicitly obtain all of the locks needed to perform a given computation. This decision was reached for two reasons. First, a system that would handle the acquisition of locks automatically would be at or beyond the state-of-the-art for database systems. Since this is not related to ArchOS' prime research goals, we would prefer not to expend the effort that such a capability would require. Second, even if we had an automatic lock acquisition facility, it seems inevitable that the system would be more prone to deadlocks than if the client explicitly requested the locks. This is due to the fact that the automatic

system would often have to upgrade a "read" lock to a "write" lock during the course of a transaction. The attempt to upgrade the lock could lead to a deadlock situation, which might have been avoided if a "write" lock had been requested in the first place. Although the automatic system would have no way of knowing that a "write" lock would eventually be needed, the client who wrote the program would indeed have known and could have avoided the situation in many cases.

The final major point to be addressed concerning locks is the use of the *ReleaseLock* primitive--specifically, a justification for the use of this primitive within a transaction.

Although the use of the *ReleaseLock* primitive within a transaction can violate the locking protocols of the transaction system, it can also be used in a manner consistent with those rules. In particular, tree locks can be released during the course of a transaction if they are not needed for the transaction computation. For example, if a given tree lock were obtained only to lock some of its descendants in the tree, then that lock could be released after the locks on the descendants have been obtained, with no undesirable effects. Also, it is possible that explicit releasing of locks might be useful in taking full advantage of Sha's notion of setwise serializability [Sha 84].

A few other minor items concerning locking issues should be mentioned. The syntax chosen for declaring locks in an arobject is similar to the syntax used in declaring variables (with types DISCRETE - LOCK - ID or TREE - LOCK - ID). However, the tree structure of the tree locks is defined dynamically by means of the *CreateLock* and *DeleteLock* primitives.

Also, we have not yet made a final decision concerning the "strength" of the connection between a lock and the data item it is associated with. If locks are closely associated with the data item, then the system can perform a number of checks to guarantee that the locks are being used properly. However, if this bond is weaker, then the client has more freedom to associate locks with more general or more abstract data items or even facilities. (For example, the client could obtain a lock on an arbitrary character string which may not correspond to any data item at the time the lock is obtained. This capability might be more difficult to provide if locks are tightly associated only with existent data items.) In this case, though, the client must use a programming convention in order to guarantee that the locks are used properly in accessing the data. These two examples are the end points of a range of possible lock strengths. We have yet to decide where in this range of possibilities the ArchOS supported locks should lie.

Finally, ArchOS does provide a client-specified timeout parameter to bound the execution time of a given transaction. This provides a crude facility to prevent deadlocks from tying up the transaction

for an arbitrarily long time. ArchOS will not necessarily detect a deadlock condition for the client; the ultimate responsibility for handling the possibility of a deadlock belongs to the client. However, ArchOS will provide some deadlock detection facilities. The exact nature of these facilities has not yet been fully determined, but some candidate facilities are: ArchOS will detect all deadlock cycles of length two (or perhaps three), or ArchOS will detect all deadlocks that involve the resources of only a single node. The level of deadlock detection service provided will be strongly influenced by the cost of providing that service. Since we will not be able to detect all deadlock conditions, we will only perform that deadlock analysis that is relatively inexpensive, while still capable of detecting some of the more common situations. In the event that a deadlock is detected, ArchOS will abort transactions as required in order to allow processing to continue without deadlock.

4.4.2.8 Rationale for Inclusion of Critical Regions in ArchOS

ArchOS supports critical regions to assure exclusive access to shared data items within a single arobject by means of the *Region* primitive. However, it may be noted that ArchOS also supports another, more secure, method of obtaining mutually exclusive access to shared data items: the transaction.

Since ArchOS is already committed to providing a transaction facility, it may not be obvious why critical regions are also supported. In fact, the main reason is that critical regions do not require all of the powerful facilities that a transaction supplies, whether they are needed or not. It was decided that no matter how efficient the ArchOS transaction facility was, it would still probably be slower than a mechanism that only provides a small fraction of the power of a transaction (even a compound transaction, which would typically involve less processing than an elementary transaction). As a result, the critical region can be used to provide a simple mutual exclusion mechanism that will often be needed when accessing shared data objects for which the failure atomic, permanent, or serializable properties of transactions are not required.

4.4.2.9 Rationale for Transaction Nesting Rules

ArchOS allows both elementary and compound transactions to be nested arbitrarily within a transaction. Such interleavings are necessary for proper system behavior. This will be demonstrated by a number of examples that point out the necessity of each type of nesting:

- In order to provide the traditional database nested transaction view, it is necessary to allow the nesting of elementary transactions within elementary transactions.
- Where failure atomicity (the property by which either all of the actions of a transaction are performed or none are) and visibility issues (as mentioned in Section 4.4.2.5) are not of prime concern, system performance can be maximized by employing nested compound transactions rather than nested elementary transactions.

- In cases where compensate routines can provide similar functionality to more traditional "undo" operations on locked data items (for example, consider the case of compensating for the allocation of a new page of memory from the operating system), it would be advantageous for performance reasons to use compound transactions nested within elementary transactions.

- Consider the case in which a compound transaction must perform a certain computation. It is certainly reasonable to believe that it might often be carried out by a set of nested elementary transactions thereby giving all of the automatic failure recovery and visibility features of nested elementary transactions to the desired computation. It is of no interest to the lower nested levels that their locks will all be released as soon as the compound transaction commits (or aborts). (Note that the compound transaction in this case acts a great deal like the top-level transaction of a set of nested elementary transactions.)

Since all of these cases seemed to be useful, it was established that ArchOS transactions could be constructed by any arbitrary mixture of the two types of transactions.

Such mixtures of the two transaction types can be quite useful in selectively passing the locks of some child transactions to a parent transaction while releasing those of other child transactions. This ability to selectively pass locks is required since the ArchOS primitives will usually contain compound transactions (in order to prevent passing system resource locks to clients), yet client elementary transactions may be built by making *Requests* for services from other client arobjects. If a *Request* were simply a compound transaction, and the computation that resulted from the *Request* execution were considered to be a child of the *Request* transaction, then none of the requestee's locks would be returned to the requestor (client) due to the nature of a compound transaction.

A mixture of elementary and compound transactions can be used in the above example to pass the requestor the requestee's locks while releasing the system resource locks at the completion of the *Request* primitive execution. (See Figure 4-4.) By implementing the *Request* primitive as an elementary transaction with several child transactions, the desired effect can be achieved. The locks obtained by the requestee's computation are automatically passed back to the requestor and, the system actions (Req1 and Req2 in the figure) are encapsulated within child compound transactions, so the requestor will not receive any of the system resource locks (since they are not returned to the *Request* primitive's highest level elementary transaction.)

4.4.2.10 Rationale for Inclusion of the AbortIncompleteTransaction Primitive

The *AbortIncompleteTransaction* primitive provides a unique capability in handling responses from child transactions.

To understand how such a primitive facility could be used, consider the following situation of a

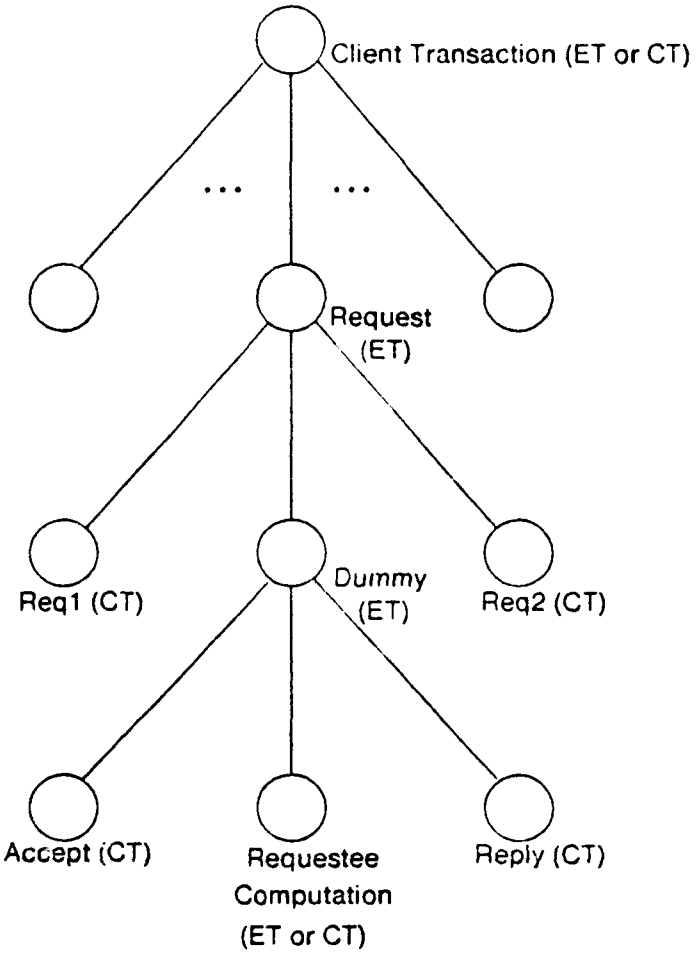


Figure 4-4: Selective Lock Passing by the *Request* Primitive

transaction performs a *RequestAll* invocation for some operation on all of the arobject instances of a certain type, then each invocation will be serviced and appropriate replies will be returned. Also, suppose that while servicing the request, each arobject instance performs some transaction processing. And finally, suppose that the requestor is only interested in the first reply to the *RequestAll* invocation. (Although this scenario may sound contrived, it is not. This is exactly the manner in which a client would attempt to obtain the shortest response time from a set of arobjects, all of which are capable of servicing the client's request. The client would simply request that all of the arobjects provide the required service (by means of the *RequestAll* primitive) and then wait until the first reply is obtained. Since, at that point, the actions of the rest of the servers are no longer of interest to the requestor, their actions can be terminated.)

After receiving the first reply, the client could not correctly proceed without the

AbortIncompleteTransaction primitive. This is due to the fact that the computations carried out by the arobject instances that respond (or would respond) after the first should be aborted (including the transaction processing that they are carrying out). However, the client has no way of knowing the identity of those arobject instances that received the request since the *RequestAll* primitive was used and therefore the exact identities of the acceptors were never explicitly known by the requestor. If the client aborts all of the processing associated with that *RequestAll* primitive, then the work done by the first replier will be lost as well. The *AbortIncompleteTransaction* primitive handles this case since it allows ArchOS to handle all of the bookkeeping associated with the identities of the acceptors and as well as performing the necessary selective aborts.

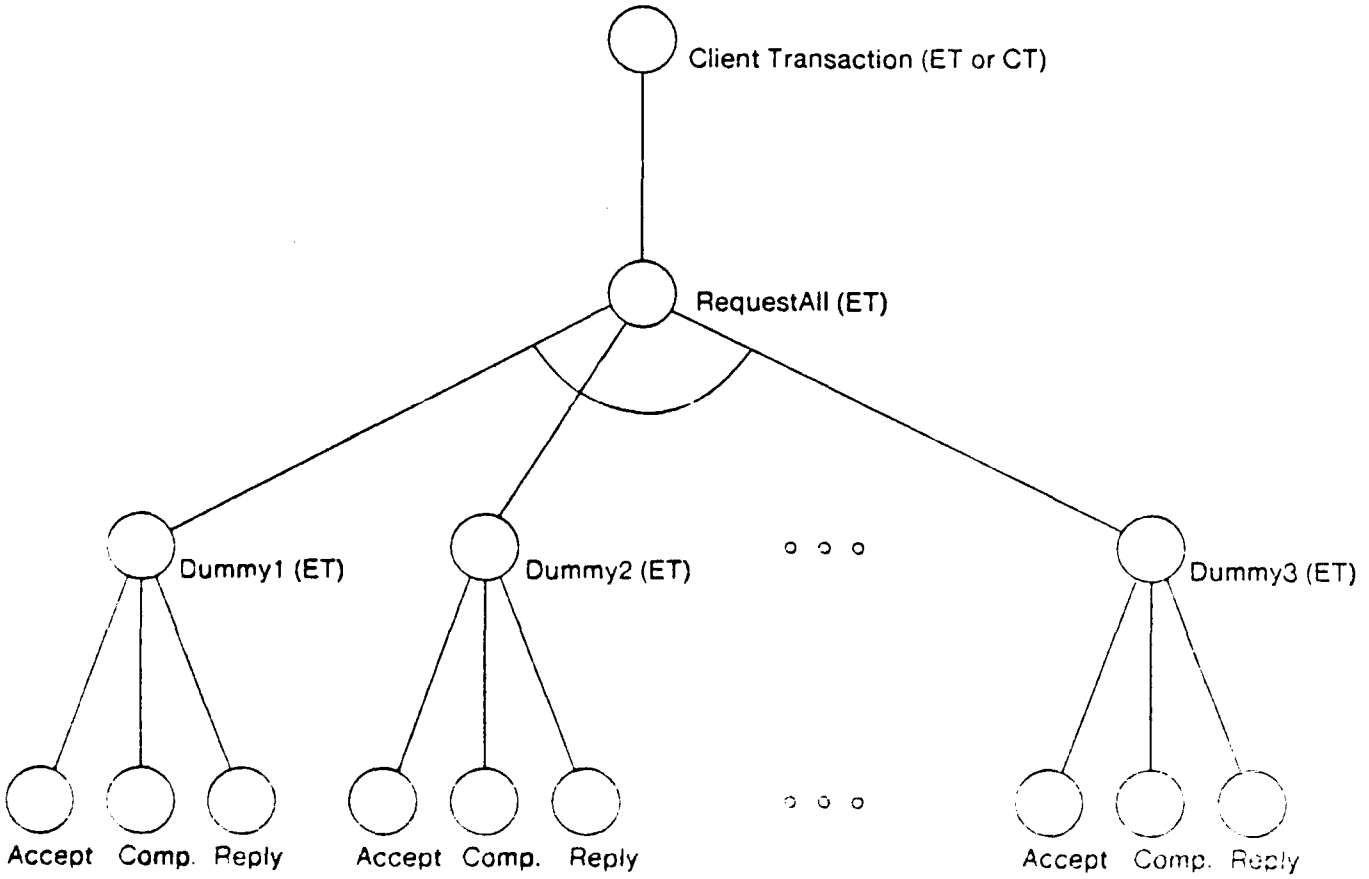


Figure 4-5: *RequestAll* Transaction Tree

Figure 4-5 shows the transaction tree structure associated with the *RequestAll* primitive. This figure will illustrate the actions taken by the *AbortIncompleteTransaction* primitive with respect to the transaction tree. The *RequestAll* transaction has several children transactions, one for each of the acceptors of the request for service. At the completion of each server's *Reply*, the subtransaction

associated with that server is concluded (committed). The *AbortIncompleteTransaction* primitive aborts all of the child transactions of the named transaction (*RequestAll*, in this case) that have not yet reached that commit point. In this way, the work that has been done will be maintained, while the incomplete work will be aborted.

This primitive has proven quite useful in example programs, including the partially replicated distributed directory example contained in Appendix A of this document.

4.4.2.11 Real-Time Facilities

ArchOS is designed to support user-defined real time deadlines, but its support for real-time systems differs fundamentally from other real-time operating systems. There are, of course, many existing real-time systems being used in a number of environments including the military, process control, and robotics. It is interesting, however, to observe some of the characteristics found in the operating systems (or *executives*, as they are sometimes called when a full operating system is not implemented) designed for these systems which provide support for real-time operation. Two primary characteristics can be described:

- These operating systems are kept simple, with minimal overhead, but also with minimal function. Virtual storage is almost never provided; file systems are usually either extremely limited or non-existent. I/O support is kept to an absolute minimum. Scheduling is almost always provided by some combination of FIFO (for message handling), priority ordering, or round-robin, with the choice made arbitrarily by the operating system.

- Simple support for management of a hardware real-time clock is provided, with facilities for periodic process scheduling based on the clock, and timed delay primitives.

Conspicuously missing from these systems at the operating system level is any specific support for managing user-defined deadlines, even though meeting such deadlines is the primary characteristic of real-time application requirements. Instead, these systems are designed to meet their deadlines by ensuring that the available resources significantly exceed the actual user requirements, and the implementation is followed by an extensive testing period to verify that this assumption is maintained under "normal" loads. In priority-driven systems, deadlines are handled by assigning a high fixed priority to processes with critical deadlines, disregarding the resulting impact to less critical deadlines.

The ArchOS primitives have been designed to provide the information which will enable the scheduling function to sequence processes according to their deadlines, ensuring that all deadlines are met in each node as long as there are sufficient resources to meet them. Techniques for handling such deadlines are well known, given sufficient resources.

An important ArchOS research area, however, is the handling of these deadlines when there are insufficient resources to handle all of them. User policies defining potential objective functions for these cases will be accepted, and a best effort will be made to implement these policies when resources are insufficient. Essentially two primary decisions are required in these cases:

- Which deadlines should be missed, and by how much? That is, should some deadlines take precedence, or should missed deadlines be fairly distributed? Should some processes be aborted if their deadlines cannot be met?

- At what point should some processes be migrated to other nodes? If so, how many processes, and which processes should be migrated?

4.4.2.12 Policy Definitions

We felt from the outset that an application running under ArchOS would need a great deal of flexibility in order to meet its specifications. Therefore, we decided to support application-defined policies to manage certain resources. At this point, we have limited the policy definition capability to two specific management tasks: process scheduling (initially on a single node) and process reconfiguration (process migration or relocation) among nodes. This decision was made because we did not feel that there was a global programming paradigm involving application-defined policies that we wished to impose on the application program writer; currently, we treat each facility on its own merits with respect to the use of application-defined policies. We do have an overall framework within ArchOS for the support of application-defined policies, but we would like to demonstrate the suitability of our ideas on a few test cases before spreading the approach throughout the entire system.

In addition, process scheduling and process reconfiguration in a distributed system are of particular interest to the real-time application programmer, who is often in the best position to determine the high-level scheduling policies of the system. This is due to the fact that the actual deadline constraints that the system must satisfy are imposed by the external environment and the nature of the application processing. By allowing the application programmer to define the scheduling policy, it is expected that the system will better be able to carry out the process scheduling task with these factors in mind, particularly in situations where there are insufficient processing resources to carry out all of the applications functions.

Another area of interest within the Archons project concerning the use of policy within a facility is the separation of the policy and mechanism portions of the facility. This work is similar in spirit (if not in implementation) to the policy/mechanism separation work done for the Hydra system [Wulf 81]. However, for the most part, ArchOS will not focus much attention on that particular aspect of policy

definition for a facility. This is partially due to the fact that, in many cases, it is very difficult to draw the line between those protions of a facility that are policies and those that are mechanisms. And since drawing that line is not a central focus of the ArchOS development effort, we feel that we should not spend a great deal of time pursuing this task. (We will indeed attempt to separate policy and mechanism insofar as possible in ArchOS, but we will not push this idea very hard.)

Once we had decided to provide support for application-defined facility policies and to make an attempt to separate policy and mechanism in the implementation of the facility, there were still a number of open questions.

First, we recognized that there is a spectrum of choices in the amount of flexibility to be provided to the application writer. One approach to the specification of a facility's services would involve: (1) determining the set of all of the policies that might ever be used to manage the facility, (2) selecting a set of mechanisms that can be combined in various ways by that set of policies to provide any of the previously identified facilities, and (3) making those mechanisms available to the application programmer to construct the facility. While this scheme allows the application writer a great deal of freedom, it is possible that it might be very difficult to actually carry out the first two steps listed above. A second approach limits the options of the application writer (and so might be more secure and reliable from the operating system's point of view) while still allowing the client to select the policy to be used in managing a facility. In this case, the application programmer would be able to select the facility policy from a limited menu of policies. (Essentially, the programmer has been given a policy mode switch.) The programmer would not necessarily even know the mechanisms that actually underlie the policies in this case. For process scheduling when processing resources are not sufficient to meet processing demands, we have chosen a scheme that lies somewhere between the extreme cases listed above. We intend to design a policy scheduling facility that has a fixed set of mechanisms that the client cannot directly access. Some of these mechanisms will evaluate value functions in order to compute an optimal (or near optimal) schedule for processes on a given node. The value functions that correspond to many of the well-known scheduling approaches will be available to the client (in a form similar to a library). However, the client may also specify the parameters for a certain class of value functions and submit a value function to the process scheduling facility to determine the exact policy to be followed. Although this approach does not allow the client to specify an arbitrary value function to the process scheduler, it does allow much more freedom than choosing from a small menu of well-known policies (such as "missing fewest deadlines" or "minimize maximum lateness").

Second, there are a number of places where application-defined policies and policy/mechanism

separation can be used. We considered the possibility that we could have two levels of application-defined policies: policies that affected the entire distributed program and policies that affected individual arobjects. We believe that there are cases where both of these alternatives make sense. However, neither process scheduling nor process reconfiguration seems to have a compelling use for arobject-level policies. As a result, the policy definition capabilities described in this document are concerned only with policies that affect the entire distributed application program. Perhaps we will pursue multiple levels of policy in a later version of ArchOS.

There is another sense in which multiple levels of policy may be discussed -- a hierarchical sense. A facility at a given level of abstract may be decomposed into a set of mechanisms that are manipulated by means of a set of policy statements. Each of the mechanisms at that level of abstraction, in turn, are also decomposable at a lower level of abstraction into a set of lower-level mechanisms manipulated by a lower-level policy; and so on. At this time, it is difficult to see exactly how such a hierarchical decomposition of facilities will aid in their design or performance. For the present, we have decided to apply the notion of policy/mechanism separation at only a single, meaningful level. (Our system was not intended to examine this policy/mechanism hierarchcy, and, once again, it is not central to the development of ArchOS as a research vehicle.)

Third, although we are currently employing only two client-defined policies, the primitives provided for the definition of policies by the application program are intended to be used for any other policies that may be defined. They are very generic in nature (associating a policy module with a special policy name and associating values with attribute names that may later be examined in carrying out policy decisions) and were intentionally selected to support a wide-range of policy definitions. We felt that a general approach to policy definition was more desirable than a specialized approach for each policy to be handled.

Finally, we have given the implementation of an application-defined process scheduling policy some thought. Since this facility must be accessed often and it must manipulate low level operating system objects (such as the queue of runnable processes), it would be advantageous for it to be resident in the kernel's memory space. We intend to have a method to allow at least that special case of an application-defined policy module be resident in kernel space. Of course, updating the process scheduling policy and coordinating the access of the policy module with client-space data will provide some additional complications, but these should not be too great.

4.4.3 ArchOS Primitives

The ArchOS primitives described in this report can be divided into two levels. The primitives which belong to the ArchOS kernel level are called *kernel* primitives and the primitives which are implemented above the kernel are called *system* primitives. This distinction will help an application designer but we did not provide any indication of which is which at this stage. The relation between the system and kernel primitives will be described in more detail in the System Architecture Specification document.

This section starts by briefly pointing out the important design problems for the complete set of ArchOS primitives. It then describes our design decisions for each primitive, reflecting the structure in the the previous chapter.

4.4.3.1 Important Design and Research Problems

In the design of the ArchOS primitives, we consider *simplicity* and *uniformity* to be the most important design goals. All of the ArchOS primitives should provide a uniform interface to a client. Thus, a primitive is accessed by a procedure (or function) invocation.

While we are trying to achieve simplicity, we also consider the primitive set's optimality as well as its completeness. Unfortunately, there is no formal notion of a complete set of primitives in a distributed operating system context [Tokuda 83a], so we have tried to evaluate the primitives' expressive power in various distributed applications.

- **Simplicity:**
  We have adopted a procedure (or function) call interface for all of our primitives. Even though the request message must be transferred to a suitable arobject by a *Request* primitive, it will be called by a procedure (or function) interface. It should be noted that our original intention was to reduce the number of primitives visible from a client process. For instance, a single *Create* primitive would be sufficient if the target language allowed "overloading" of procedures/functions. The current *CreateArobject* and *CreateProcess* could be united and used as follows:

  ```
  arobject-id = Create(arobj-name);
  process-id = Create(process-name);
  ```

  One weak point is the lack of a notion of default parameters. For example, the default value of the node-id argument in the *CreateProcess* primitive could be set to "ANY-NODE", so that a caller would not need to specify this optional argument every time. However, the current ArchOS interface requires that the client must specify all parameter values explicitly at calling time. (due to the limitation of C language.)

- **Uniformity:**
  We have provided a uniform syntax as well as uniform (i.e., network transparent or location independent) semantics for each primitive. For instance, communication

primitives provide the identical semantics for local and remote communications. Arobject and process management primitives also provide uniform semantics for creating and destroying an instance.

- **Optimality and Completeness:**
  While attempting to minimize the number of primitives in ArchOS, we paid careful attention to assure that the full expressive power of the primitive set was maintained. On the other hand, many primitives were added to support new facilities such as transaction and policy management. Also due to some language constraints, we needed to increase the number of primitives a little, but the current primitive set can be used to construct an extremely diverse set of distributed applications.

4.4.3.2 Arobject/Process Management

We view an arobject as a basic module for embodying a distributed abstract data type. In particular, we decided to treat an arobject as an *active* system entity rather than a *passive* entity. There are many advantages and disadvantages related to this decision. First, we can easily define an autonomous module. It is easy for an arobject not only to initialize its computational state by itself, but also to recover its computational state by itself. In other words, by having a single INITIAL process in each arobject, a designer can give responsibility for recovery to the INITIAL process. Second, unlike traditional procedure invocation in abstract data types, a caller cannot invoke an operation of an arobject in a *master-slave* manner, but must use a form of rendezvous. The receiver therefore has the right to accept, reject, or delay the requested function. Finally, the degree of parallelism within an arobject can be dynamically changed in many ways. For instance, a process can be created in a different node to perform a requested computation on demand or many processes can be pre-created to accept a particular type of request.

We provided a completely network transparent arobject/process management. That is, creation and destruction of arobjects and processes can be performed without knowing the location of the target arobject or process.

The following functionalities are not adopted for the current ArchOS:

- **Multiple, simultaneous creation of arobjects and processes**
  This might be useful to instantiate a replicated arobject simultaneously. However, it creates more conflict with other optional arguments such as node-id. Thus, it must be invoked once for each arobject or process instance in the current system.

- ***Hierarchical dependency among arobjects***
  There are no hierarchical dependency among arobjects except for nested arobjects in a distributed program. Thus, a single arobject can be created and destroyed with no effect on the rest of the external the arobjects in the same distributed program. However, it should be noted that nested arobjects are killed when enclosing arobject is killed.

- **Inheritance relationship among arobjects**
  There are no inheritance relationship among arobjects. In modern programming languages, such as Smalltalk-80 [Goldburg 83], Flavor [Weinreb 81], and LOOPS [Bobrow 81], the inheritance relationships among objects have shown great advantages, improving the structural sharing among abstract objects and simplifying the modification of the existing objects. We considered a multiple inheritance relationship among arobjects, but there were many difficult problems in providing a way to forward a request message to its super(class) arobject. In our arobject paradigm, we wished to avoid an unnecessarily (deep) hierarchy among arobjects; thus we did not adopt the inheritance relationship. However, the current model can support a nested arobject which is private to the outer arobject.

- **Hierarchical dependency among cooperating processes**
  There is no hierarchical relationship among cooperating processes. For instance, in ADA [Ada 83], the termination of a task depends upon the termination of its inner blocks' tasks. Thus, the termination of a higher-level task may be delayed. In Shoshin [Tokuda 83b], every process must belong to a family tree which indicates such termination dependencies. A process in Shoshin can be attached to or detached from the creator's family tree at creation time. This dependency information might be useful during the debugging phase, but the current system does not support this facility. That is, a process can kill other processes or terminate itself without causing any additional killing among communicating processes.

There are some restrictions on the use of the *Kill* primitive in ArchOS. First, a process can kill only a process which exists in the same arobject. The reason is that, in principle, a process should not be able to "see" within the body of the other arobjects. Even if a process were visible from the outside of its arobject, that process should be protected from being killed by other arobjects in order to protect its own environment.

4.4.3.3 Communication Management

The primitives for communication management were designed to provide support not only for a conventional client-(single)server model but also for cooperation among multiple servers in a distributed environment. Thus, the system supports a *Request-Accept-Reply* type of communication among cooperating arobjects in either a synchronous or an asynchronous manner.

To support such a cooperating server model, we also created a *one to many* type of communication among cooperating arobjects. That is, a process can invoke an operation on a group of particular arobject instances at once. To provide this type of communication, we have adopted asynchronous *Request* (i.e., *RequestAll* and *RequestSingle*) and *GetReply* primitives. Unlike a completely synchronous communication, the requestor does not need to wait for all replies to come back. However, this asynchronous feature may increase the complexity of transaction control.

As for various server structures, we are interested in a collection of servers which can increase the availability and reliability of service. The productivity of the servers may also increase together in a collection of cooperating workers. Even for the single server case, it is easy to make an arobject autonomous and cooperative, since there is at least one process managing its service. Unlike the master-slave relationship in remote procedure calls, the server can control not only the sequence of incoming messages, but also the execution order of the actual services.

The current communication management does not support or adopt the following functionality:

- **Message forward primitive**
  A message forward primitive was not added in order to maintain the basic rule that only the arobject receiving the request message can send a reply to the caller. Furthermore, within an arobject, a process can simulate a forwarding function by creating a new process and passing the message as its initial parameter. The lack of a general forward primitive may adversely affect the creation of pipelined server processes, but it prevents illegal use of transactions across the arobject boundary.

- **Remote procedure call paradigm**
  Although a process can communicate with other arobjects only by invoking arobject operations, no master-slave relation exists in our message passing paradigm. Our model insists on a rendezvous type of communication among arobjects. Note that when a process accesses a private data object from a remote node, a conventional remote procedure call will be used. If such a data object and the calling process are located in the same node, then the normal procedure call will be used with identical semantics to the RPC case.

- **Built-in timeout facility in communication primitives**
  In order to bound the execution time of communication activity, we considered adding one more argument, namely a timeout value, to each communication primitive. However, we preferred to give the user a timeout facility outside the communication primitives. It should be noted that each communication primitive may contain a compound transaction so that it is also bounded internally.

4.4.3.4 Synchronization

There are two levels of synchronization support in ArchOS. One is a critical region for controlling shared private data objects and the other is explicit locking primitives for controlling inter-transaction activities.

There are many problems with the critical region construct. For instance, if a process dies within a critical region, the state of the shared objects cannot easily be restored. It is also difficult to bound the total waiting time as well as the execution time of the critical region.

Despite these difficulties, a critical region scheme was adopted, since we expect that creating a

compound transaction for such a shared object may be unnecessary in certain situations. For instance, there are cases in which permanency of data objects is sufficient without providing full failure atomicity.

The explicit lock and release primitives were necessary to control concurrent transaction activities. It is clear that by using a discrete lock type, we could encounter a deadlock. In case of a simple deadlock, ArchOS might detect and resolve it; however, in general, it will remain the user's responsibility to detect and resolve deadlock problems.

On the other hand, by using a tree lock protocol a user can avoid the deadlock problem in some cases. However, a user must declare the dependency among locks in a tree by using the *CreateLock* primitive. (See Section 4.4.2.7).

The current synchronization management does not support or adopt the following functionality:

- **Error recovery in a critical region scheme**
  We believe that any shared object which requires failure atomicity should be accessed from a transaction. For instance, we can use a compound transaction to replace the critical region by using the CT{ ... } construct.

- **Timeout (exception) handling within a critical region**
  The critical region construct simply takes a timeout parameter to bound the total execution time of the caller. This timeout mechanism does not provide a corresponding exception handler. Thus a user must provide its function.

4.4.3.5 Transaction/Recovery Management

A compound or elementary transaction construct creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone. However, there are several restrictions on these steps needed to maintain the basic properties of transactions. These restrictions are as follows:

- Between any two transactions, a transaction cannot pass any its computational state to the other transaction by using a non-atomic data object. The following Example 1 shows two illegal transaction scopes in a single process.

Example 1: Computational State Passing via a Non-atomic Data Object

```
Arobject Server body
{ ...
  process INITIAL(parameters)
  {     . . .
        NormalType State1, State2;    /* Non-Atomic objects */

        ET1(timeout){
            State1 = Compute1(arg1, ..., argk)
        }
        . . .
        ET2(timeout){
            State2 = Compute2(State1, arg1, ..., argk)
        }
  }
}
```

In Example 1, the illegal (state) passing occurs within the INIT process by using the non-atomic data object, $State_1$. Since ET_2 depends on the value of $State_1$, the transaction property of ET_2 will not maintained.

- From two independent transaction scopes, two arobjects cannot communicate with each other by using a communication primitive or a shared abstract data object. This is an example of so-called *cooperating transactions* [Sha 83] and the current ArchOS model does not support it.

  For instance, the following two examples show normal and cooperating transaction scopes.

Example 2: A Normal Transaction Scope

```
Arobject Requestor body
{ ...
  process INITIAL(parameters)
  {     . . .
        ET(timeout){
            Request(Server-Aid, SERVICE, reqmsg, repmsg);
            . . . steps . . .
        }
  }
}

Arobject Server body
{ ...
  process INITIAL(parameters)
  {     . . .
        while (true) {
                t-opr-aid = AcceptAny(ANYOPR, reqmsg);

                CT(timeout){   /* begin compound transaction */
                    . . . steps . . .
                    do_service(t-opr-aid.opr, reqmsg, replymsg);
                }   /* end transaction */

                Reply(t-opr-aid.tid, replymsg);
        }
  }
}
```

Example 3: A Cooperating Transaction Scope

```
Arobject Requestor body
{ ....
  process INITIAL(message)
  {     . . .
        ET(timeout){
            Request(Server-Aid, SERVICE, reqmsg, repmsg);
            . . . steps . . .
        }
  }
}

Arobject Server body
{ ...
  process INITIAL(message)
  {     . . .
        while (true) {
                CT(timeout){  /* begin compound transaction */
                     t-opr-aid = AcceptAny(ANYOPR, reqmsg);
                     . . . steps . . .
                     do_service(t-opr-aid.opr, reqmsg, replymsg);
                     Reply(t-opr-aid.tid, replymsg);
                }    /* end transaction */
        }
  }
}
```

In Example 2, it is easy to determine the execution sequence of the two transactions. That is, ET in the requestor happened before CT in the server. On the other hand, Example 3 shows two concurrent arobjects, namely two INITIAL processes, communicating with each other from within two separate transactions.

- If an elementary transaction contains a nested compound transaction, there is a possibility of deadlock due to the nature of the lock management. The following example shows the possibility of deadlock within a single transaction.

Example-4: A Deadlock within a Single Transaction

```
Arobject Server body
{ ...
  Process Server(message)
  {     . . .
        ET(timeout) {  /* begin elementary transaction */
            . . . ET step 1 . . .
            sval = SetLock(DISCRETE, Lx, WRITE) /* get a lock on ob
            . . . operate on X . . .

            CT(timeout){  /* begin compound transaction */
                . . . CT steps . . .
                /* get a lock on object X */
                sval = SetLock(DISCRETE, Lx, WRITE)
                <<. . . operate on X . . .>>  /* deadlock! */
            }   /* end transaction */
            . . . ET step 3 . . .
        }   /* end transaction */
}
```

The problem occurs when the nested compound transaction tries to obtain a lock on object X. Since the lock was already taken by the top level transaction ET, this compound transaction will abort due to timeout.

The current transaction/recovery management facility does not support or adopt the following functionality:

- **Cooperating transactions**
  Cooperating transactions are not supported as explained above.

- **Detection of deadlock in the lock management**
  Complete detection in the locking facility is not provided. A detection mechanism, in general, was left for the application designer. However, the proper use of tree-locks may help the client to avoid deadlock.

- **Automatic generation of a compensate action for a compound transaction**
  It is very complicated to automatically generate a compensate action for each operation involved by a compound transaction. Thus, this was left for the application designer. ArchOS supports only the automatic execution of well-defined compensate actions when a compound transaction is aborted.

- **Automatic lock management for a shared object**
  To determine the proper set of locks and their locking rules from the program is not an easy task, and often the amount of sharing obtained is limitied due to simpleminded locking rules. The lock management is also left for the application designer.

4.4.3.6 File Management

The current file management in ArchOS supports a single-level file structure and does not provide any directory structure for users. However, the system can provide at least three kinds of file properties. First, a *normal* file has the data portion of the file arobject in volatile storage. Second, a *permanent* file's data portion is allocated in permanent (non-volatile) storage. Finally, an *atomic* file has the data portion which is declared as an atomic data object.

Since there is no notion of file protection at this level, a protection domain can be build based on the scope of the reference name.

The current file management does not support or adopt the following functionality. (Note that the following features are not permanently removed from ArchOS. These functionalities may be easily adopted on top of the current file arobject's structure).

- **Directory structure**
  There is no directory structure in the current file system. A client must use a flat file name space to classify files.

- **A replicated atomic file type**
  This replicated file type must vary in terms of its access protocols and replication schemes. Thus, there is no system supported replicated file type so far.

- **File sharing and protection scheme**
  There is no conventional access list for a file object, rather each file will have a set of locks to control concurrent file accesses.

It should be noted that if a normal or permanent file is accessed from a transaction scope, all of the transaction properties will not be provided. The file must be an atomic file type in order to fully utilize the transaction facility.

4.4.3.7 I/O Device Management

The I/O device management provides a set of access primitives for normal or special devices in the system. These primitives are similar to the ones which are used for file management, but ArchOS does not provide complete compatibility between the two.

The current I/O device management does not support or adopt the following functionality.

- **Full compatibility between devices and files**
  The main reason was that it is difficult to cover all different, specialized devices as standard (i.e., atomic, permanent or normal) files.

- **Transaction facilities are not supported on the *real* devices**

Unlike arobjects, a user cannot create a transaction for a sequence of device operations. This is because "undo"s are often impossible following real actions taken on these devices.

4.4.3.8 Time Management

The time management primitives provide functions for obtaining getting time information and/or setting/resetting scheduling parameters for performing time-driven scheduling. This time-driven scheduling is based on ArchOS' best effort scheduling model. Since this model requires at least *request, delay, deadline*, and *estimated execution times* as basic parameters, the *Delay* and *Alarm* primitives were designed to provide these parameters from a client process.

There are several approaches to the provision of these functions in a real time operating system, but these were chosen in an attempt to provide maintainability and modularity as well as time control. For example, many systems provide a process time-out setting to be made at which an application process will be scheduled. This frequently requires breaking a module at the delay point into two processes, rendering the processing difficult to read and understand. The *Delay* and *Alarm* primitives provide for such delays to be coded inline in the application without breaking the process into multiple processes.

At the same time, the deadline-driven scheduling algorithm to be used in ArchOS is provided with the time parameters needed to apply deadline policies. A part of the research being conducted on the Archons project concerns process scheduling in the presence of application-defined deadlines. It is intended that application-defined policies will be used to control deadline-driven scheduling which will attempt to meet deadlines or minimize the damage if insufficient resources are available to meet deadlines.

The current time management does not support or adopt the following functionality:

- **Asynchronous Delay or Alarm primitive**
  Since we would like to avoid interrupt-driven timeout routines, the Delay and Alarm primitives are blocking primitives. This reduces the complexity required in the implementation of a process body. To provide this type of asynchronous handling, the user must create a new (time) process to avoid the blocking.

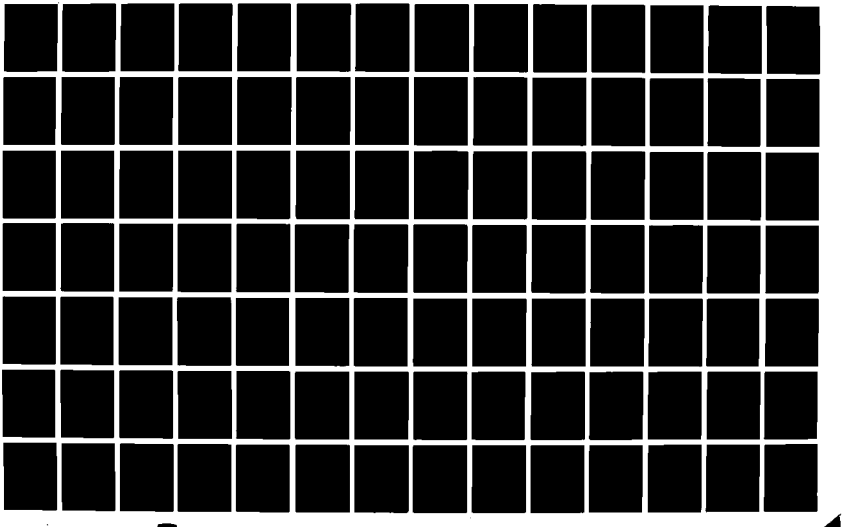
4.4.3.9 Policy Management

The policy management primitives provide functions which manipulate the policy module. The policy set arobject maintains a set of policy modules. Each policy module represents a real policy and maintains its own

AD-A184 401

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
CONCEPTS & TECHNIQUES FOR SUPPORT OF REAL TIME DISTRIBUTED OPERATING SYSTEMS
BY: CARNEGIE-MELLON UNIVERSITY

2 OF 3
NOSC TD 1113
UNCLASSIFIED
JUL 87



The current policy management does not support or adopt the following functionality:

- **Dynamic addition/deletion of a new user-defined policy module**
  The current model can only support predefined set of user policies, thus a new user-defined policy cannot be recognized by ArchOS.

4.4.3.10 System Monitoring and Debugging Support

The system monitoring and debugging facilities provide various abilities to control the behavior of cooperating arobjects and their processes during execution. All of the primitives for system monitoring and debugging can only be used by a specially privileged arobject, so that a normal client arobject cannot issue any of them.

The current system monitoring and debugging facilities do not support or adopt the following functionality:

- **Single-step function for tracing process activity**
  A single step function is not supported yet, but may be added in the future.

- **Creating an arbitrary break point in an arobject**
  A client cannot set an arbitrary break point in an arobject. The available breaking points are only the communication points where a process sends or receives a message. A monitoring process may trap the following execution step by using the CaptureComm or WatchComm primitives.

- **Specialized debug functions such as *Redo* or *Undo* operation for arbitrary functions**
  These facilities should be built by using the current facilities, rather than creating a set of new primitives.

4.5 Program Examples

This appendix presents an example problem involving a distributed data object along with two potential solutions employing arobjects and the ArchOS system primitives.

4.5.1 The Problem: A Distributed Directory with Partially Replicated Data

In this example, the problem is to implement a distributed directory, where the directory data is physically spread among several processing nodes with each directory entry replicated at multiple nodes in order to improve reliability. Notice that there is no guarantee that every node has a complete copy of the directory information.

In particular, the problem is to construct a distributed directory of some fixed length, where each entry in the directory associates a name (a string of characters) with a value (in the examples, the

values used are assumed to be integers). The data in the directory must be partially replicated at several different processing nodes, and directory users must always have a consistent view of the directory.

This type of distributed directory represents a class of data object that will be common in an ArchOS client's distributed program---an object that is highly robust (due to the presence of multiple copies of each critical data item at physically separate nodes) but does not require that the user of this data object be aware of the physical distribution involved in the implementation. (In fact, this implementation is invoked by the user in exactly the same manner as a centralized data object.)

4.5.2 The General Approach to the Problem

The solutions presented in the next two sections are both built upon the same fundamental approach. The directory data is contained in several different arobjects (called *directory-copy* arobjects), located at different processing nodes. These arobjects each contain some portion of the total directory state, but it is unlikely that any two of them contain exactly the same data or that any one of them contains all of the current directory data.

Another entity is provided to consistently access and maintain the directory-copy arobjects so that the directory user "sees" a single directory object (called the *directory* arobject). This entity takes a different form in the two solutions that follow, and this is their major distinguishing feature. In the first example, one or more directory arobjects accept directory operation invocations and then service them by making the appropriate invocations on the directory-copy arobjects. All of the arobjects in this solution are separate entities, and so this solution views the organization of arobjects as being quite "flat."

In the second solution, the entire directory service is provided by a single arobject. This arobject is physically distributed over several nodes and has placed multiple processes throughout the system to accept directory operation invocations. In addition, the directory-copy arobjects are embedded within the directory arobject as private arobjects in the second solution. As a result, this solution views arobjects as being organized in a hierarchical manner.

ArchOS supports both views of the organization of arobjects (flat or hierarchical). Neither of them appears to be better than the other for this example, but the client is free to organize arobjects in the manner that seems most advantageous for the application at hand.

Both solutions manage the coordination of the multiple directory copies in the same way: by

applying the notion of using only a quorum [Gifford 79, Herlihy 84] of directory copies in order to carry out any operation on the distributed directory. Using this method allows the directory to continue operating smoothly even if several directory copies are unavailable at any given time. Each directory operation (lookup, add, delete) requires only a quorum of directory copies be involved to perform that operation. In order to determine the latest value of a given name, the directory-copy arobjects also maintain a version number with each entry name. (The current value of a given name corresponds to the entry for that name with the highest version number in any directory copy.)

Also, notice that the transaction facility has been used in order to guarantee the consistency of the directory information at all times. Elementary transactions have been used for the most part since the directory operations invoked by a user of the directory arobject may be part of a larger transaction, and in that case, if compound transactions had been used, the consistency of the directory could no longer be guaranteed.

4.5.3 Solution 1: Two Cooperating Classes of Arobjects

As outlined in the previous section, this solution uses several separate directory and directory-copy arobjects distributed throughout the system. The code for these arobject types is assumed to be contained in two files: "directory.arb" contains the code for the directory arobject and "dir-copy.arb" contains the code for the directory-copy arobject.

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "directory.arb"           *
*                                                                *
*****************************************************************/

#define DIR-COPIES              2*N+1
#define READ-QUORUM             N+1
#define WRITE-QUORUM            N+1
#define DEL-QUORUM              2*N+1
/*  If a DEL-QUORUM cannot be formed, NULL can be written to the value
        field of a new version of thedesired name.  Later, when all
        copies are available, a delete can again be attempted.  */

arobject directory specification
{
    char *name;
    int value;
    STATUS result;

    operation add(name, value) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value);
}

arobject directory body
{
    ***** insert message type declarations here *****

  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type DCNAME = {
    permanent AROBJ-REFNAME dir-copy-name;

  /*  Procedures to Operate on Atomic Data Items  */

    procedure store(dcname)
    AROBJ-REFNAME dcname;
    {
        dir-copy-name = dcname;
    }

    AROBJ-REFNAME function get()
    {
        return(dir-copy-name);
    }
  }     /*  end of private-abstract-data-type DCNAME  */

  DCNAME dcname;

    process INITIAL(dircopyname)
    AROBJ-REFNAME dcname;
    {
        MESSAGE *requestmsg;
        OPERATION opr;
```

```
    dcname.store(dircopyname);

    while(TRUE) {
        AcceptAny(ANYOPR, requestmsg);
        opr = msg-header-operation(requestmsg);
        CreateProcess(opr, requestmsg);
    }
}

process add(add-requestmsg)
{
    char *name;
    int value;
    int i;
    STATUS rd-result;
    int val[READ-QUORUM];
    int version[READ-QUORUM];

    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid, trans-id;

    PID pid;
    struct add-replymsg {
        STATUS result;
    }

    strcpy(name, requestmsg.body.name);
    value = requestmsg.body.value;

    pid = msg-header-caller(add-requestmsg);
    tid = msg-header-tid(add-requestmsg);

    ET(timeout) {
        trans-id = SelfTid();

        read-quorum(name, val, version, rd-result);
        if (rd-result != OK) {
            Reply(pid, tid, {FAIL});
            AbortTransaction(trans-id);
        }
        max-version = -1;
        for (i=1; i<= READ-QUORUM; i++)
            max-version = max(max-version, version[i]);
        write-quorum(name, value, max-version+1, wr-result);
        add-replymsg.body.result = wr-result;
        Reply(pid, tid, add-replymsg);
    }
    if (IsAborted(trans-id)) {
            /* handle error condition here */
    }
}

procedure read-quorum(name, value, version, result)
char *name;
int value[READ-QUORUM];
```

```
    int version[READ-QUORUM];
    STATUS result;
    {
        int count;

        TRANSACTION-ID tid;
        FIND-REPLYMSG find-replymsg;

        count = 0;
        /*  initiate timeout  */
        tid = RequestAll(dcname.get(), find, {name});
        while (count < READ-QUORUM) {
            GetReply(tid, find-replymsg);
            if (find-replymsg.body.result == OK) {
                count++;
                value[count] = find-replymsg.body.value;
                version[count] = find-replymsg.body.version;
            }
        }
        /*  Need to fix case where quorum cannot be established.  */
        AbortIncompleteTransaction(tid);
        result = OK;
    }

    procedure write-quorum(name, value, version, result)
    char *name;
    int value, version;
    STATUS result;
    {
        int count;

        TRANSACTION-ID tid;
        struct add-replymsg ...

        count = 0;
        /*  initiate timeout  */
        tid = RequestAll(dcname.get(), add, {name, value, version})
        if (add-replymsg.body.result == OK) count++;
        while (count < WRITE-QUORUM) {
            GetReply(tid, add-replymsg);
            if (add-replymsg.body.result == OK) count++;
        }
        /*  Need to fix case where quorum cannot be found.  */
        AbortIncompleteTransaction(tid);
        result = OK;
    }
}       /*  End of directory arobject body  */
```

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "dir-copy.arb"            *
*                                                                *
*****************************************************************/

arobject directory-copy specification
{
    char name[MAXSIZE];
    int value;
    int version-no;
    status result;

    operation add(name, value, version-no) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value, version-no);
}

arobject directory-copy body
{
  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type TABLE = {
    TREE-LOCK-ID t-lock[TABLESIZE];
        /* define tree lock structure as a "linear" tree, such that
                t-lock[1] is the root and has child t-lock[2];
                t-lock[2] has child t-lock[3];
                and so on. */
    atomic struct table[TABLESIZE] {
        char *name;
        int value;
        int version;
    }

  /*  Procedures to Operate on Atomic Data Items  */

    procedure init-table()
    {
        int i;
        TIME timeout = TIMEOUT;
        TRANSACTION-ID tid;

        /*  define tree lock structure  */
        t-lock[1] = CreateLock(NULL-LOCK-ID);
        for (i=2; i<=TABLESIZE; i++) {
            t-lock[i] = CreateLock(t-lock[i-1]);
        }

        CT(timeout) {
            tid = SelfTid();
            for (i=1; i<=TABLESIZE; i++) {
                SetLock(TREE-LOCK, t-lock[i], WRITE);
                table[i].name = NULL;
                ReleaseLock(TREE-LOCK, t-lock[i], WRITE);
            }
        }
```

```
    if (IsAborted(tid)) {
        /* handle error condition */
    }
}

procedure lookup(name, index)
char *name;
{
    int index, i;

    TIME timeout = TIMEOUT;

    index = -1;
    ET(timeout) {
        for (i=1; i<=TABLESIZE; i++) {
            SetLock(TREE-LOCK, t-lock[i], READ);
            if (strcmp(table[i].name, name) == NULL) {
                index = i;
                breakfor;
            }
            ReleaseLock(TREE-LOCK, t-lock[i], READ);
        }
    }
}

STATUS function enter(index, name, value, version-no)
char *name;
int index, value, version-no;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
        tid = SelfTid();

        SetLock(TREE-LOCK, t-lock[index], WRITE);
        if (table[index].name != NULL || table[index].name != name)
            AbortTransaction(tid);
        strcpy(table[index].name, name);
        table[index].value = value;
        table[index].version = version-no;
    }
    if (IsCommitted(tid)) return(OK);
    else return(FAIL);
}

procedure get-fields(index, name, value, version-no, result)
char *name;
int index, value, version-no;
STATUS result = FAIL;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
```

```
            tid = SelfTid();

            SetLock(TREE-LOCK, t-lock[index], READ);
            strcpy(name, table[index].name);
            value = table[index].value;
            version-no = table[index].version;
        }
        if (IsCommitted(tid)) result = OK;
    }
}   /*  end of private-abstract-data-type TABLE    */

TABLE tab;

  process INITIAL()
  {
      OPERATION opr;

      tab.init-table();

      while (TRUE) {
          AcceptAny(ANYOPR, requestmsg);
          opr = msg-header-operation(requestmsg);
          CreateProcess(opr, requestmsg);
      }
  }

  process add(add-requestmsg)
  ADD-REQUESTMSG add-requestmsg;
  {
      char *name;
      int value, version-no;
      int index;
      STATUS stat, enter();
      TIME timeout = TIMEOUT;

      struct add-replymsg {
          STATUS result = FAIL;
      }

      strcpy(name, add-requestmsg.body.name);
      value = add-requestmsg.body.value;
      version-no = add-requestmsg.body.version;

      ET(timeout) {
          index = lookup(name);
          if (index < 0) index = lookup(NULL);
          if (index < 0) AbortTransaction(SelfTid());
          stat = enter(index, name, value, version-no);
          if (stat != OK) AbortTransaction(SelfTid());
          add-replymsg.body.result = OK;
      }
      Reply(msg-header-caller(add-requestmsg),
              msg-header-tid(add-requestmsg), add-replymsg);
  }
```

```
    process find(find-requestmsg)
    FIND-REQUESTMSG find-requestmsg;
    {
        char *name;
        int index, value, version-no;
        char *entry-name;
        STATUS stat;

        struct find-replymsg ...;
        PID pid;
        TRANSACTION-ID tid;

        strcpy(name, find-requestmsg.body.name);
        pid = msg-header-caller(find-requestmsg);
        tid = msg-header-tid(find-requestmsg);

        index = lookup(name);
        if (index < 0) Reply(pid, tid, {NOT-FOUND});
        else {
            get-fields(index, entry-name, value, version-no, stat);
            if (strcmp(name, entry-name) == NULL)
                Reply(pid, tid, {OK, value, version-no});
            else Reply(pid, tid, {FAIL});
        }
    }
}       /*  End of directory-copy arobject body  */
```

4.5.4 Solution 2: A Single, Distributed Arobject

As previously discussed, this solution implements the entire directory server as a single arobject in which several directory-copy arobjects have been included as private arobjects. As in the previous solution, the code for the directory arobject is located in "directory.arb," while the code for the directory-copy arobject is located in "dir-copy." In fact, the code for the directory-copy arobject is exactly the same as in the first solution; however, the code for the directory arobject has been changed somewhat. The greatest change has taken place in the INITIAL process of the directory arobject; also, the directory arobject now contains a statement that identifies the description of the directory-copy arobject as a private arobject.

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "directory.arb"           *
*                                                                *
*****************************************************************/

#define DIR-COPIES              2*N+1
#define READ-QUORUM             N+1
#define WRITE-QUORUM            N+1
#define DEL-QUORUM              2*N+1
/*  If a DEL-QUORUM cannot be formed, NULL can be written to the value
        field of a new version of thedesired name.  Later, when all
        copies are available, a delete can again be attempted.  */

arobject directory specification
{
    char *name;
    int value;
    STATUS result;

    operation add(name, value) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value);
}

arobject directory body
{
    ***** insert message type declarations here *****

  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type DCNAME = {
    permanent AROBJ-REFNAME dir-copy-name;

  /*  Procedures to Operate on Atomic Data Items  */

    procedure store(dcname)
    AROBJ-REFNAME dcname;
    {
        dir-copy-name = dcname;
    }

    AROBJ-REFNAME function get()
    {
        return(dir-copy-name);
    }
  }     /*  end of private-abstract-data-type DCNAME  */

  DCNAME dcname;

    private-arobject directory-copy = "dir-copy.arb";

    process INITIAL()
    {
        NODE node[TOTAL-NODES] = {NODEA, NODEB, ... ,NODEN};
        int i;
```

```
    AID aid;

    for (i=1; i<=COPIES; i++) {
        aid = CreateArobject(directory-copy, NULL, node[i]);
        BindArobjectName(aid, "dir-copy");
    }
    dcname.store("dir-copy");
    for (i=1; i<=SERVERS; i++)
        CreateProcess(accept-invocs, NULL, node[i]);
}

process accept-invocs()
{
    MESSAGE *requestmsg;
    OPERATION opr;

    while(TRUE) {
        AcceptAny(ANYOPR, requestmsg);
        opr = msg-header-operation(requestmsg);
        CreateProcess(opr, requestmsg);
    }
}

process add(add-requestmsg)
{
    char *name;
    int value;
    int i;
    STATUS rd-result;
    int val[READ-QUORUM];
    int version[READ-QUORUM];

    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid, trans-id;

    PID pid;
    struct add-replymsg {
        STATUS result;
    }

    strcpy(name, requestmsg.body.name);
    value = requestmsg.body.value;

    pid = msg-header-caller(add-requestmsg);
    tid = msg-header-tid(add-requestmsg);

    ET(timeout) {
        trans-id = SelfTid();

        read-quorum(name, val, version, rd-result);
        if (rd-result != OK) {
            Reply(pid, tid, {FAIL});
            AbortTransaction(trans-id);
        }
        max-version = -1;
```

```
        for (i=1; i<= READ-QUORUM; i++)
            max version = max(max-version, version[i]);
        write-quorum(name, value, max-version+1, wr-result);
        add-replymsg.body.result = wr-result;
        Reply(pid, tid, add-replymsg);
    }
    if (IsAborted(trans-id)) {
        /* handle error condition here */
    }
}

procedure read-quorum(name, value, version, result)
char *name;
int value[READ-QUORUM];
int version[READ-QUORUM];
STATUS result;
{
    int count;

    TRANSACTION-ID tid;
    FIND-REPLYMSG find-replymsg;

    count = 0;
    /*  initiate timeout  */
    tid = RequestAll(dcname.get(), find, {name});
    while (count < READ-QUORUM) {
        GetReply(tid, find-replymsg);
        if (find-replymsg.body.result == OK) {
            count++;
            value[count] = find-replymsg.body.value;
            version[count] = find-replymsg.body.version;
        }
    }
    /*  Need to fix case where quorum cannot be established.  */
    AbortIncompleteTransaction(tid);
    result = OK;
}

procedure write-quorum(name, value, version, result)
char *name;
int value, version;
STATUS result;
{
    int count;

    TRANSACTION-ID tid;
    struct add-replymsg ...

    count = 0;
    /*  initiate timeout  */
    tid = RequestAll(dcname.get(), add, {name, value, version})
    if (add-replymsg.body.result == OK) count++;
    while (count < WRITE-QUORUM) {
        GetReply(tid, add-replymsg);
        if (add-replymsg.body.result == OK) count++;
```

```
            }
            /*  Need to fix case where quorum cannot be found.  */
            AbortIncompleteTransaction(tid);
            result = OK;
        }
    }       /*  End of directory arobject body  */
```

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "dir-copy.arb"            *
*                                                                *
*****************************************************************/

arobject directory-copy specification
{
    char name[MAXSIZE];
    int value;
    int version-no;
    status result;

    operation add(name, value, version-no) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value, version-no);
}

arobject directory-copy body
{
  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type TABLE = {
    TREE-LOCK-ID t-lock[TABLESIZE];
        /* define tree lock structure as a "linear" tree, such that
                t-lock[1] is the root and has child t-lock[2];
                t-lock[2] has child t-lock[3];
                and so on. */
    atomic struct table[TABLESIZE] {
        char *name;
        int value;
        int version;
    }

  /*  Procedures to Operate on Atomic Data Items  */

    procedure init-table()
    {
        int i;
        TIME timeout = TIMEOUT;
        TRANSACTION-ID tid;

        /*  define tree lock structure  */
        t-lock[1] = CreateLock(NULL-LOCK-ID);
        for (i=2; i<=TABLESIZE; i++) {
            t-lock[i] = CreateLock(t-lock[i-1]);
        }

        CT(timeout) {
            tid = SelfTid();
            for (i=1; i<=TABLESIZE; i++) {
                SetLock(TREE-LOCK, t-lock[i], WRITE);
                table[i].name = NULL;
                ReleaseLock(TREE-LOCK, t-lock[i], WRITE);
            }
        }
```

```
    if (IsAborted(tid)) {
        /* handle error condition */
    }
}

procedure lookup(name, index)
char *name;
{
    int index, i;

    TIME timeout = TIMEOUT;

    index = -1;
    ET(timeout) {
        for (i=1; i<=TABLESIZE; i++) {
            SetLock(TREE-LOCK, t-lock[i], READ);
            if (strcmp(table[i].name, name) == NULL) {
                index = i;
                breakfor;
            }
            ReleaseLock(TREE-LOCK, t-lock[i], READ);
        }
    }
}

STATUS function enter(index, name, value, version-no)
char *name;
int index, value, version-no;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
        tid = SelfTid();

        SetLock(TREE-LOCK, t-lock[index], WRITE);
        if (table[index].name != NULL || table[index].name != name)
            AbortTransaction(tid);
        strcpy(table[index].name, name);
        table[index].value = value;
        table[index].version = version-no;
    }
    if (IsCommitted(tid)) return(OK);
    else return(FAIL);
}

procedure get-fields(index, name, value, version-no, result)
char *name;
int index, value, version-no;
STATUS result = FAIL;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
```

```
            tid = SelfTid();

            SetLock(TREE-LOCK, t-lock[index], READ);
            strcpy(name, table[index].name);
            value = table[index].value;
            version-no = table[index].version;
        }
        if (IsCommitted(tid)) result = OK;
    }
}   /*  end of private-abstract-data-type TABLE    */

TABLE tab;

  process INITIAL()
  {
      OPERATION opr;

      tab.init-table();

      while (TRUE) {
          AcceptAny(ANYOPR, requestmsg);
          opr = msg-header-operation(requestmsg);
          CreateProcess(opr, requestmsg);
      }
  }

  process add(add-requestmsg)
  ADD-REQUESTMSG add-requestmsg;
  {
      char *name;
      int value, version-no;
      int index;
      STATUS stat, enter();
      TIME timeout = TIMEOUT;

      struct add-replymsg {
          STATUS result = FAIL;
      }

      strcpy(name, add-requestmsg.body.name);
      value = add-requestmsg.body.value;
      version-no = add-requestmsg.body.version;

      ET(timeout) {
          index = lookup(name);
          if (index < 0) index = lookup(NULL);
          if (index < 0) AbortTransaction(SelfTid());
          stat = enter(index, name, value, version-no);
          if (stat != OK) AbortTransaction(SelfTid());
          add-replymsg.body.result = OK;
      }
      Reply(msg-header-caller(add-requestmsg),
              msg-header-tid(add-requestmsg), add-replymsg);
  }
```

```
        process find(find-requestmsg)
        FIND-REQUESTMSG find-requestmsg;
        {
            char *name;
            int index, value, version-no;
            char *entry-name;
            STATUS stat;

            struct find-replymsg ...;
            PID pid;
            TRANSACTION-ID tid;

            strcpy(name, find-requestmsg.body.name);
            pid = msg-header-caller(find-requestmsg);
            tid = msg-header-tid(find-requestmsg);

            index = lookup(name);
            if (index < 0) Reply(pid, tid, {NOT-FOUND});
            else {
                get-fields(index, entry-name, value, version-no, stat);
                if (strcmp(name, entry-name) == NULL)
                    Reply(pid, tid, {OK, value, version-no});
                else Reply(pid, tid, {FAIL});
            }
        }
    }       /*  End of directory-copy arobject body  */
```

5. ArchOS System Architecture Description

This chapter describes the system architecture of ArchOS which supports all of the facilities and primitives described in the client interface document [Jensen 85]. Since we view "ArchOS system architecture" as a high-level view of the internal system design, this chapter focuses on the description of the major components of ArchOS operating system and leaves the detailed design to the next chapter.

From a conceptual point of view, ArchOS consists of the ArchOS kernel and system arobjects which are grouped together to form *subsystems*. The ArchOS kernel provides a set of basic mechanisms which can support arobjects in both kernel and client address spaces. An ArchOS subsystem consists of one or more system arobjects which provides an ArchOS facility. In other words, the ArchOS primitives described in the client interface document are defined as a set of operations of these system arobjects.

5.1 Overview

ArchOS is not a time-sharing operating system nor a network operating system for a set of workstations. ArchOS is a physically dispersed operating system which performs decentralized system-wide resource management for a *decentralized computer* which can be physically dispersed across 10^{-3} to 10^{3} meters, interconnected without the use of shared primary memory.

We view ArchOS as a reserach vehicle to perform research on the issues of decentralization in real-time distributed systems at the OS level and below. Thus, the ArchOS facilities were designed to allow test applications to be constructed with which to study ArchOS characteristics, but they need not provide a particularly complete set of facilities in an application production environment. As we specified in the client interface document, some system facilities are not fleshed out, but each facility is functionally closed so that each function can be evaluated with respect to our research issues.

The basic model for system-wide resource management in ArchOS is based on our previous conceptual/theoretical study and experimental study. In particular, we are interested in decentralized resource management schemes where each global decision is made *multilaterally* by a group of peers through negotiation, compromise, and consensus. Thus, ArchOS facilities are provided by a group of cooperating servers of which each service entity is built by an arobject on each node. Since each arobject can have its own state and can activate computation independently of other arobjects, cooperation among servers can be easily represented by a set of arobjects.

Furthermore, an arobject cannot fetch or modify the status of any other arobject without invoking its corresponding operation at the target arobject, thus the modularity of arobjects can also be achieved as well as potential parallelism between the caller and called arobjects.

From the higher-level point of view, the overall structure of ArchOS is divided into two system components. One component is an ArchOS kernel which provides a set of basic mechanisms and links the higher-level policy with the mechanisms and creates an arobject environment. The other is a set of system arobjects which implements ArchOS system facilities. A system arobject can be further distinguished by where it resides, namely a kernel arobject which exists in the kernel and non-kernel arobjects which exist outside the kernel.

Since an ArchOS facility is built based on cooperating arobjects, it is easy to change the internal implementation without impacting client programs. Furthermore, certain facilities are also built based on the notion of policy/mechanism separation [Wulf 81], so that an existing policy can be changed or a new type of policy can be created without changing mechanisms. Thus, ArchOS facilities can be easily tuned towards a particular type of application environment without changing the mechanisms in the system.

5.2 ArchOS System Architecture

The system architecture of ArchOS should be flexible enough so that various degrees of decentralized resource management schemes can be applied in ArchOS without incurring major modification cost. Since the Archons project is not attempting to develop a computing facility, we must consider "openess" of the system architecture.

The system structure of ArchOS was designed to improve the robustness, extensibility, and modularity of OS functions. There is no centralized decision maker in the system for any type of system-wide resource management. That is, any system-wide resource management is performed by the cooperation among the essentially identical peer server modules at each node. These modules allow the use of the "most decentralized" algorithms (that are practical) for managing system resources.

The ArchOS system architecture was designed without assuming any specific hardware-level structures such as node architecture and communication architecture. However, to maximize performance, it is preferable to have the following architectural support: 1) Each node should have two or more processors with a reasonable amount of shared memory; 2) Each node can communicate

with another node, a set of nodes (i.e., multicasting), or all of the nodes within a reasonable amount of time; 3) Each node should have a logically homogeneous environment. That is, even if the processor architectures are different, all of the nodes can emulate the ArchOS functionality.

5.2.1 Objectives

The major objectives are derived from the research requirements and the functionality of ArchOS. In particular, ArchOS is designed to be a research vehicle for investigating various decentralized resource management techniques, so that various types of OS facilities can be added, changed or deleted.

The design objectives of ArchOS system architecture are summarized as follows:

- **Open system architecture:**
  Since ArchOS is designed as a research vehicle to investigate various decentralization issues at operating system level and below, a new system component can be easily added as well as deleting the existing facility.

- **Highly decentralized resource management:**
  ArchOS should not have any centralized decision maker in the system and should be structured as essentially identical peer modules replicated at each system node.

- **High system availability:**
  To maintain high system availability in the face of node and communications faults, provide for no lasting degradation of system function or response beyond the actual resource loss encountered.

- **High system modularity:**
  Previous distributed systems have been constructed around the physical communications limitations of the system, as opposed to being based on such modern software engineering principles as abstract data types and information hiding for defining modularity.

- **Highly extensible facilities:**
  To construct highly extensible facilities which will limit the cost of redesign for implementing widely divergent operating system facilities for experimentation with the fundamental concepts of interest to us in this research.

- **Reasonable system performance:**
  To provide reasonable system performance to support applications with real-time constraints. ArchOS must consider time-critical functions for which time constraints define some system failure modes (i.e., issues of timeliness will not be ignored as they are in some systems, nor will they be dealt with by simply providing a large ratio of available to currently used resources, which is how virtually all other real-time systems strive to meet response time requirements).

5.2.2 Basic Approach

A basic approach to meet the previous objectives is that we provide a uniform view of the system components, namely a kernel and a set of arobjects. An ArchOS kernel provides basic mechanisms for ArchOS facilities and the cooperating arobjects support the policies to provide a particular set of services. While we avoid a large monolithic kernel, we try to improve robustness, modularity, and extensibility of each ArchOS facility as well as increasing potential parallelism by using the cooperating arobjects.

To meet the previous objectives, we will use the following approaches:

- **Open system architecture:**
  We will build ArchOS based on an object-oriented architecture view, so that the ArchOS kernel is not a monolithic kernel and consists of a set of basic mechanisms and kernel arobjects. On top of the kernel, ArchOS facilities are provided by cooperating arobjects. Since an arobject encapsulates its associated information (information hiding), key properties of each facility can be easily modified without affecting the other facilities. Furthermore, additional system flexibility will be provided by identifying and separating mechanism and policy in ArchOS facilities.

- **Highly decentralized resource management:**
  Since we try to avoid having any type of centralized decision maker for system-wide resources and each arobject can have its own computational state and knowledge, it is easy to form cooperation among peer modules in the system. Unlike traditional procedure invocation in abstract data types, a caller cannot invoke an operation on an arobject in a *master-slave* manner, but must use a form of rendezvous. Based on its own state and decision, the receiver arobject has the right to accept, reject, or delay the invocation of the requested operation.

- **High system availability:**
  The use of atomic transactions to maintain consistency will result in continuance of useful computation in the presence of lost, delayed, inaccurate, or incomplete information. Also, a service handled by a set of replicated arobjects can be easily supported by using the one-to many communication capability together with transactions.

- **High system modularity:**
  Since an arobject can encapsulate its state and implementation details from the other arobjects, it is easy to use it as a system building block. System modularity is also achieved not only by using arobjects, but also through policy/mechanism separation in ArchOS facilities.

- **Highly extensible facilities:**
  ArchOS facilities are implemented as a set of system arobjects, thus it is possible to create a new type of service by creating a new set of arobjects and registering them.

- **Reasonable system performance:**
  To provide reasonable system performance, ArchOS subsystems are built based on a set

of arobjects which can be executed on a multiprocessor based node without major redesign.

A conceptual view of the ArchOS system architecture is depicted in Figure 5-1. On each node built from one or more processors with shared memory, there are ArchOS kernel and system arobjects.

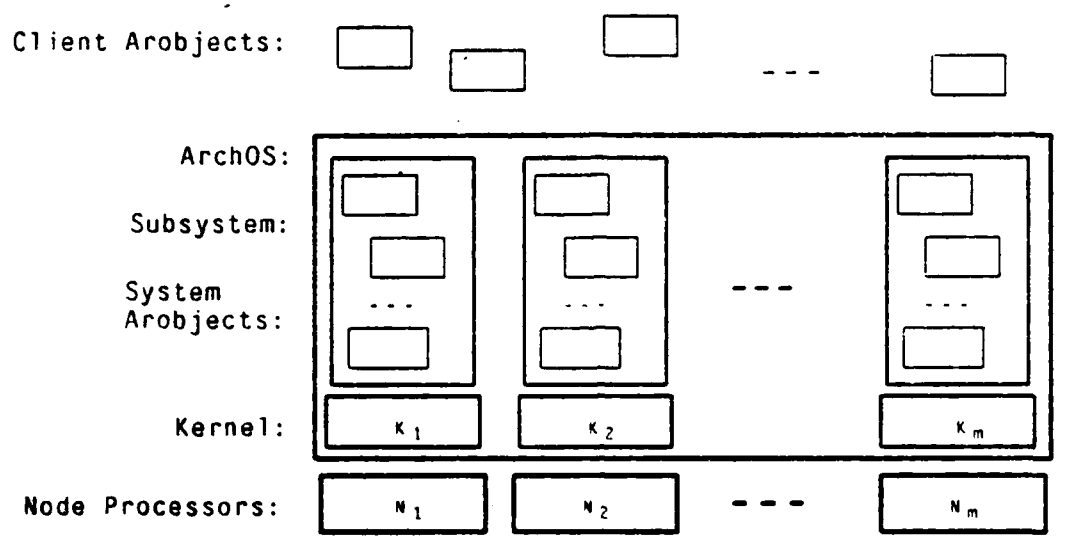


Figure 5-1: Overview of ArchOS System Architecture

The ArchOS *kernel* consists of a set of basic mechanisms and a set of system arobjects which reside in a kernel domain. No system arobjects can be accessed without invoking the ArchOS primitives. A client process must use a communication primitive, *request*, to invoke any system primitives. ArchOS primitives defined in the client interface are provided either as *kernel* primitives or *system* primitives. Kernel primitives are defined as an operation on kernel arobjects, while system primitives are provided by an operation on the system arobject which exists above the kernel.

ArchOS Subsystems are one or more arobjects grouped together by sharing the same responsibility for providing a particular *service*, namely a set of functions in an ArchOS facility.

A *system arobject* can be a kernel arobject or an ordinary arobject which resides in the user domain. A *client arobject* can be created on top of the kernel and resides in the user domain.

5.2.3 Functional Dependency among ArchOS Subsystems

The relationship among ArchOS subsystems can be summarized by abstracting out the detailed interaction among arobjects across the various subsystems. Since each subsystem consists of a set of cooperating arobjects, many activities are initiated by invoking a particular operation in the arobject. This section focuses on the functional dependency among ArchOS subsystems, rather than on the detailed interaction among arobjects. The description of the internal structure of each subsystem will be given in the next chapter.

From a functional point of view, ArchOS subsystems can be depicted as in Figure 5-2.

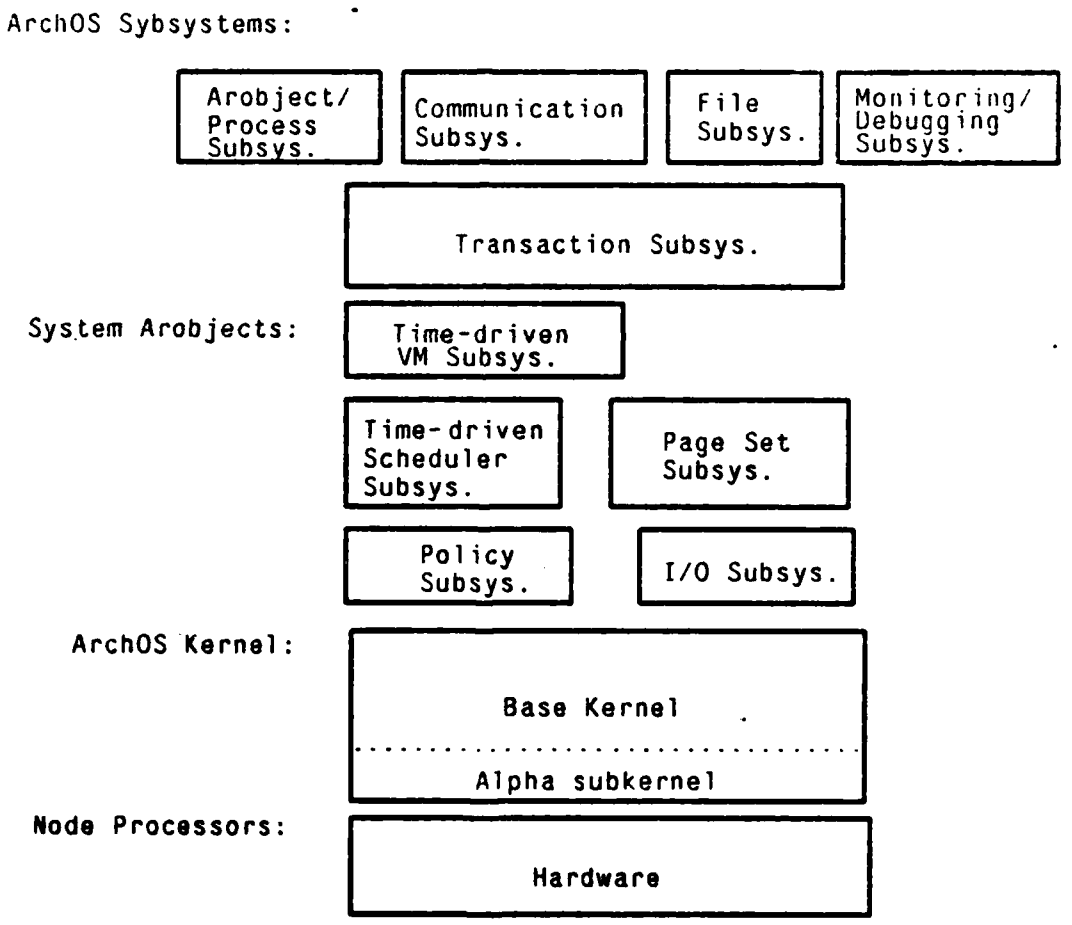


Figure 5-2: Functional Dependency among ArchOS Subsystems

In Figure 5-2, each box represents a subsystem in ArchOS. However, unlike a traditional hierarchical layered structured system, an operation at a lower-level subsystem can be directly invoked from a client arobject.

The *ArchOS Kernel* provides a set of basic mechanisms and consists of a set of system arobjects which reside within a kernel domain. Within the kernel, there are two subkernel layers: the lower-level is called the *Alpha* subkernel, and the higher-level is called the *Base* kernel. Unlike a traditional monolithic kernel, the ArchOS kernel contains a set of of system arobjects, which we refer to as *kernel arobjects*, and it can initiate independent computation from client arobjects. A certain set of basic mechanisms are realized to coordinate with the policy definition module, so that facilities may provide a different type of functionality to clients.

The *Policy management subsystem* maintains a user-definable system module, called the *policy definition module* which consists of a *policy body* and a set of *policy attributes*. Since a policy definition module can be placed in the kernel, a system arobject, or a client arobject, the policy management subsystem creates and destroys a *policy definition descriptor* which indicates the location of the policy definition and the information related to the policy body and attributed set.

The *Page set subsystem* provides a uniform view of secondary storage, namely a *page set* in ArchOS. An arobject can access a file through the page set and logical disk subsystems. On each node, there is a page set manager which can allocate or deallocate a permanent type or atomic type of page set on a specified logical disk.

The *I/O device subsystem* manages interaction between I/O devices and arobjects. Any signal from I/O devices is handled at this subsystem and is translated into an invocation request for a system arobject.

The *System monitoring and debugging subsystem* provides various abilities to monitor and control the behavior of cooperating arobjects and their processes during execution.

The *Time-driven scheduler subsystem* provides a time-driven scheduling facility which may be altered by adding, changing or selecting a different type of scheduling policy. The basic mechanism is based on ArchOS' "best effort" scheduling model [Locke 85]. This subsystem also provides functions such as obtaining time information and setting/resetting scheduling parameters for clients.

The *Time-driven virtual memory subsystem* provides a virtual memory management facility in a time-driven fashion. The time-driven scheduler may be called from this subsystem to initiate pre-paging in/out activities in such a way that the system can execute time-critical tasks in a timely manner.

The *Transaction subsystem* provides transaction mechanisms for operations on arbitrary types of

arobjects. This subsystem supports two types of transactions, namely *elementary* and *compound* transactions which can be nested in arbitrary combinations. To coordinate an atomic update for a transaction, the transaction subsystem must cooperate with one or more page set managers which the transaction has visited.

The *Arobject/process management subsystem* provides the basic functions to create and destroy arobjects and processes. This subsystem also manages binding and unbinding of arobjects/processes *reference* names and supports *binding protocols* to match a requestor with one or more suitable server arobjects. The requestor arobject invokes an operation by using the *invocation protocol* through the communication subsystem.

The *Communication subsystem* provides the *invocation protocol* and manages arobject invocation among arobjects. This subsystem provides not only *one-to-one* communication for a conventional client-(single) server model, but also *one-to-many* communication for the client-cooperating multiple server model.

The *File subsystem* provides system-wide location-independent file access. Although the file subsystem does not support a hierarchical file name space, a system-wide flat name space is maintained at each node. The file subsystem also provides three kinds of file properties. A *normal* file has the data portion of the file arobject in volatile storage. A *permanent* file's data portion is allocated in permanent (non-volatile) storage and an *atomic* file has its data portion declared as an atomic data object. For atomic files, the transaction facility is also supported through the transaction subsystem.

5.3 Structure of ArchOS Subsystems

An ArchOS subsystem consists of a set of system arobjects, called *components*, resident on each node and provides a system-wide or local service within a node. Each subsystem was designed based on a generic server structure for ArchOS in order to provide reliable and fast service of the requested system functions.

5.3.1 Objectives

The internal structure of a subsystem should meet the following objectives:

- Various types of decentralized resource management schemes should be easily implemented in a subsystem.
- A subsystem should be able to serve concurrent requests efficiently, fairly and without deadlocks.

- A subsystem should be able to support replicated arobjects to increase availability of its service.
- Each component should be able to stop independently and resume its activity without major disturbance.
- Within a component, concurrent operation invocation should be easily provided if the service requests can be handled simultaneously.

5.3.2 Internal Structure of Subsystems

The internal structure of subsystems is based on a generic server model which can provide system-wide service in a reliable manner. The server model was used to provide the template for different types of servers and to reduce the design complexity of subsystems.

The server model consists of *service protocols*, which define a complete interaction between a client and all servers providing the same service, and a *server structure*.

Our view of system service at the subsystem level is depicted in Figure 5-3.

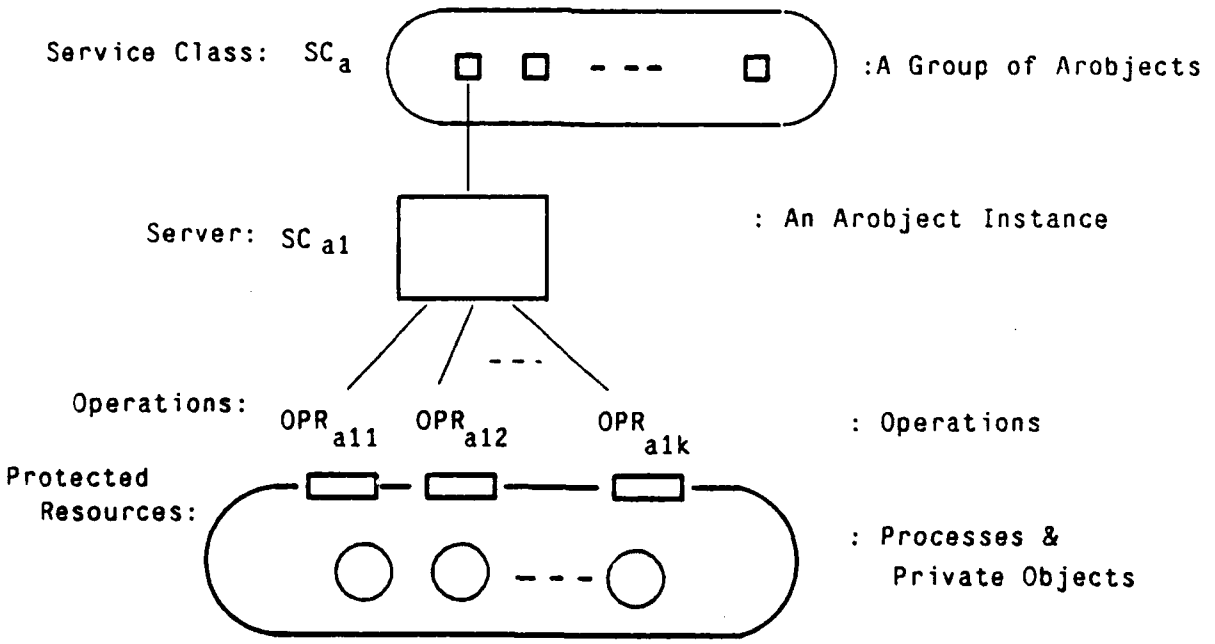


Figure 5-3: Relationship among Service, Server, and Arobject

At the subsystem level, a *reference name* is given to each *service class* which provides a particular service, namely a set of functions for an ArchOS facility. A service class is implemented by a set of service instances, namely *servers*. A server is formed by an instance of an arobject and is normally replicated at each node. However, a different set of servers can also form a service class, since a reference name can be shared by these servers.

In general, services are replicated on each node, and system-wide resource management is performed by using the service protocol to define interaction between client and server as well as inter-server communication.

5.3.2.1 Service Protocol

A subsystem provides a service or a set of services for a client. A client can locate a suitable server for a requested service and get the result back by initiating the service protocol. The service protocol consists of three protocols: *binding*, *invoking*, and *inter-server* protocols.

The *binding* protocol defines how a client determines a suitable server for a requested service from a set of servers. For instance, we can apply the following binding policies for certain types of resource management such as creating a new instance of an arobject or a process in the system:

- **First Fit (FF):**
  The first server from the matched servers will be selected to submit a service request.
- **Random Fit (RF):**
  A server from the random selection will be used to submit a service request.
- **Best Fit (BF):**
  One of the best matched servers will be used to submit a service request.
- **Best Effort Fit (BEF):**
  The best matched server according to the best-effort (decision making) algorithm will be used to submit a service request.

It should be noted that RF protocol can be used without any interaction with potential servers, while FF protocol may need at least one reply from the servers, BF needs all replies, and BEF needs all or some replies.

The *invoking protocol* defines how a client can invoke an operation at a specific or non-specific server(s) and can get the service result. In ArchOS, a client arobject can invoke an operation synchronously as well as asynchronously. Thus, it is possible for a client to invoke an operation at multiple servers simultaneously.

- **synchronous/specific server request:**
  A client can invoke an operation on a specific arobject by using a *Request* primitive.

- **synchronous/non-specific servers request:**
  A client can invoke an operation on a specific service class by using a *RequestAll* primitive followed by the necessary *GetReply* primitives.

- **asynchronous/specific server request:**
  A client can invoke an operation on a specific arobject with blocking itself by using a *RequestSingle* primitive.

- **asynchronous/non-specific servers request:**
  A client can invoke an operation on a specific service class by using a *RequestAll* primitive.

The *Inter-server protocol* defines how an individual server can exchange control or status information to perform a service. Thus, the inter-server protocol is dependent upon the nature of the service. For instance, local resource information of a server can be distributed to all other servers periodically by invoking a suitable operation for a service class. This type of multicasting can also be performed by using a *requestall* primitive.

5.3.2.2 Generic Structure for a Server

The generic structure for a server is based on the nature of service the server provides and the characteristics of an arobject. Since any system service should be highly available, a server itself must increase its availability by avoiding necessary blocking periods and deadlocks.

Each arobject has at least one process which has responsibility to create the necessary task force to provide a service. The number of processes may be created dynamically depending on the request load. On the other hand, a certain server might have a fixed number of processes to handle a fixed number of tasks. Thus, the server structure is related to the nature of the service which is provided by the cooperating arobjects.

We can summarize the generic structure for a server in Figure 5-4.

- Type I: *a single process*
  The initial process accepts, requests, and processes them one at a time. That is, the execution order of the operations will be totally serialized. Each operation invocation will be handled by a corresponding procedure or function within the initial process.

- Type II: *functionally partitioned workers*
  The initial process creates a number of workers according to the number of operations. Coordination between workers is handled by the workers. Each operation invocation is accepted by a specific worker process assigned to do the task. Thus, a worker issues the *Accept* primitive to wait for a specific type of requested operation.

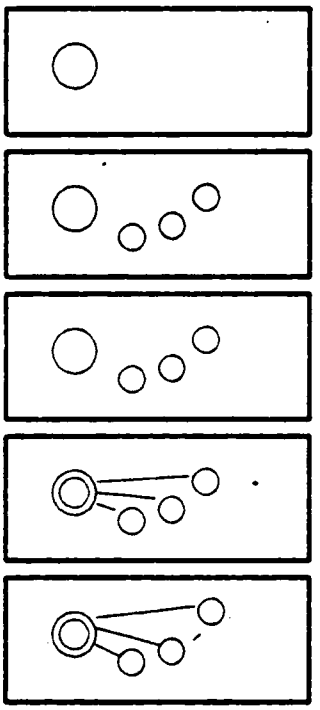


Figure 5-4: Generic Server Structures for a Server

- Type III: *replicated workers*
  The initial process creates a number of replicated workers to accept "any" operation. Coordination between workers is handled by the workers. Workers can be placed at different nodes, so parallel execution of requested operations might be possible. Each worker issues an *AcceptAny* primitive to wait for an invocation request.

- Type IVa: *a manager with a fixed number of workers*
  The initial process creates a fixed number of workers as a service task force and becomes a manager of them. Coordination between workers is handled by the initial process and each worker may or may not be assigned the same task.

- Type IVb: *a manager with variable number of workers*
  The initial process acts as a manager of workers and accepts all incoming invocation requests and creates necessary workers on demand. Coordination between workers is handled by the initial process and each worker may or may not be given the same functionality.

6. ArchOS System Design Description

This chapter contains the ArchOS system design description and describes the internal architecture of each ArchOS subsystem. From a conceptual point of view, the ArchOS operating system consists of the ArchOS kernel and a set of subsystems composed of a set of cooperating system arobjects. While the ArchOS system architecture description in the previous chapter discussed higher-level views of the system structure, this chapter focuses on how each subsystem is built based on cooperating arobjects.

6.1 Overview

The ArchOS system design description discusses the internal architecture of ArchOS focusing on the ArchOS kernel and subsystems. The ArchOS kernel provides basic mechanisms for ArchOS facilities and each subsystem implements each service facility of ArchOS.

As discussed in the previous chapter, the ArchOS kernel consists of two layers: the lower layer is the *Alpha* subkernel and the higher layer is the ArchOS base kernel. On top of the base kernel, a kernel arobject can be managed to increase the concurrency within the base kernel. ArchOS provides a system-wide arobject environment on top of the ArchOS kernel. Since ArchOS facilities are implemented by cooperating system arobjects, they can be modified, added, or deleted without having a major cost.

A subsystem consists of a set of cooperating arobjects which can provide the actual services to clients. In general, each *component* of the subsystem is replicated on each node, so a system-wide service is provided as the result of interaction among components. Each component also has the responsibility to recover from so-called "clean and soft failure" [Bernstein 83] at a node.[3] The chosen recovery sequence maintains the consistent state of the server.

In the following sections, we describe the ArchOS kernel and each ArchOS subsystem focusing on its internal structure, key algorithms, and inter-server protocols.

---

[3] We limit our discussion of resiliency against "clean and soft" failure in which some of nodes of the system simply stop running and loose the contents of main memory, but the contents of stable storage used by the failed nodes (computers) remain intact.

6.2 ArchOS Kernel

6.2.1 Overview of Kernel

The ArchOS Kernel is a virtual machine that provides basic service mechanisms necessary to support the ArchOS operating system. The abstractions provided at the kernel interface are not all be implemented directly, however. Thus, the kernel as presented to clients is built by projecting components of the kernel's internal structures and enhancing those projections as necessary to achieve the client interface.

Internally, the kernel has a logical structure that is in three levels. The lowest level, called the *executive* level, interfaces directly to the hardware and provides mechanisms to control physical resources and some internal kernel resources on a per-node basis. These mechanisms include management of physical memory, processor dispatching, virtual memory, communications, process management, and primitive objects.

Above the executive is the *subkernel*. The subkernel implements some support required for clients that is not required within the executive level, and it provides primitive support for inter-machine coordination. In particular, it supports client synchronization primitives and primitive transaction coordination. The current implementation of the subkernel for ArchOS is known as *Alpha*.

The subkernel supports the *kernel* layer, which then implements the client interface. The kernel layer contains support for compound transactions, and support for arobjects. The kernel layer also provides such access as is necessary for clients to subkernel and executive services. For example, a client's request to create a process will result in the kernel layer making a request on the executive layer that a process be dispatched from a pool of waiting processes. The pool is not visible to clients, however. The view at the client level is of a new process being created.

Figure 6-1 shows a diagram of the kernel structure.

6.2.2 ArchOS Alpha Subkernel

The Alpha subkernel will be described in terms of its major functions. The logical diagram above presents basic dependencies: higher level functions use lower level ones. A system as complex as ArchOS does not always exhibit such a clearly hierarchical set of dependency relations, however. Thus, the diagram should not be viewed as describing all allowable interactions. The subkernel's functions will be described from the lowest levels, moving up.

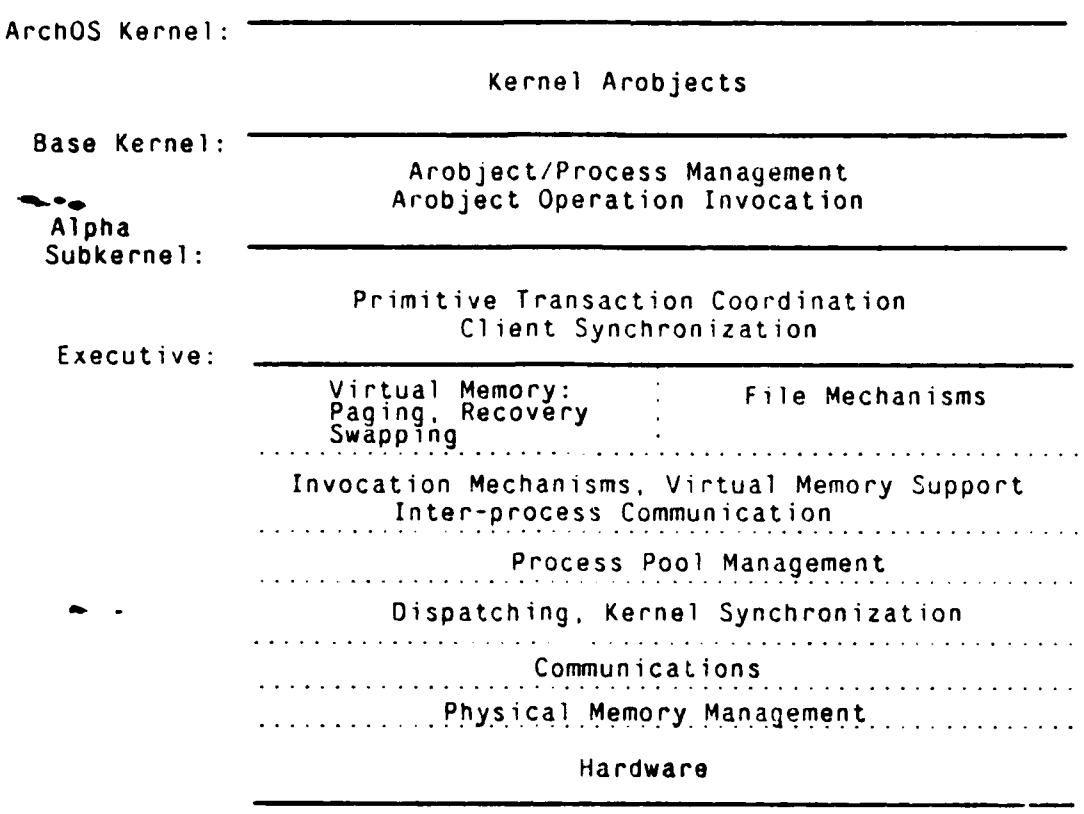


Figure 6-1: Logical Structure of ArchOS Kernel

Hardware:

The Alpha subkernel is designed for a wide class of machines. The first implementation of the kernel requires that machines have standard virtual memory with fixed size pages, and may have one or several processors. Duties are assigned to processors on a functional basis. For example, one processor might handle client processing, one might be assigned to basic operating system processing, and one to inter-node communication.

Physical Memory Management:

Two sets of services are provided within the kernel to manage physical memory. The most basic mechanism provides a pool of pages. These pages can then be used for client processes, objects or for specific kernel functions. The number of free pages at any time serves as input to the paging system, which will page out portions of client processes and objects if the number of free pages is ever insufficient. Thus, at any time it is possible to allocate a page quickly in response to a demand.

A second mechanism controls allocation of memory in portions that are smaller than a page. This is called the *kernel heap*. The kernel heap must use the page allocation mechanism. Areas from the kernel heap are used to contain basic kernel data structures, such as object control blocks and message headers.

Communications:

Within the kernel, simple communication services offer direct communication with other nodes and, where applicable, access to network services such as broadcast. These services are on a best-effort basis. The kernel's communication services will transmit a message, for example, but will not take any action to ensure its arrival. Higher level software, such as the transaction coordination and interprocess communication mechanisms, must use specific techniques for retransmission, duplicate suppression, and other such requirements.

In multiprocessor nodes, the communications mechanisms permit receipt of messages on the basis of *logical* criteria. Thus, it is possible, for example, to transmit a message addressed to a *transaction*, instead of to specific nodes. If a node's communication system recognizes that the transaction has visited the node, the message is passed to the operating system processor. Otherwise, it is discarded.

Kernel Process Support:

Within the kernel, a pool of unassigned processes is maintained. Each such process, called a *kernel process*, is partially constructed. Kernel processes are dispatched for several purposes. For example, a kernel process may be used to handle an incoming invocation, to control the virtual memory system, or it may *assume* a *client process* identity. This last case occurs in response to a request by a client to create a process.

Several specific mechanisms are required to support kernel processes. Conventional mechanisms are used to switch process contexts and dispatch processes. Low-level synchronization mechanisms are provided: semaphores to control mutual exclusion, and memoryless events to indicate occurrence of repeating conditions. One such event is used to indicate changes in the state of the unassigned process pool. When this event is caused, a maintenance process is awoken. If the pool is low, this process will create more processes for the pool; if the pool is too high, some processes from the pool will be discarded. This mechanism permits fast dispatch when processes are needed by performing pool maintenance in the background.

Invocation Mechanisms, Interprocess Communication:

Each component of a client's system, whether process or object, is assigned an independent address space. Interactions between components may require modification of the address space to make parameters and messages accessible to their recipient. Furthermore, such data must be examined by the kernel, to determine their destination. A set of mapping mechanisms support these requirements by permitting the kernel to map and remap pages between address spaces and into the kernel's address space.

Invocation of operations on primitive objects is accomplished by modifying the invoking process's address space to make the object's data accessible. The invoking process then executes code within the object. In the case that the object is not resident on the same machine as the invoking process, a process is dispatched from the pool on the machine containing the object, and it is this process that represents the calling process during the invocation.

Messages may be transmitted between system components by executing *send* and *receive* operations at the level of the client interface. These operations are then translated by the kernel software into modifications to the address spaces of the communicating processes and, where necessary, communication between nodes supporting the communicating processes. Support for this is provided in the subkernel's virtual memory mapping mechanisms.

Virtual Memory:

A system component's virtual address space can exceed the amount of available memory. Also, objects may not be accessed for long periods. Paging, swapping, and primitive file system mechanisms support the necessary abstractions. Although it is not visible to an ArchOS client, a process or an object may have components of its address space, or its entire state, written to secondary storage.

Although there is no specific need for direct client access to a file system in an object system, support is provided to permit a client to modify portions of its address space. Thus, it is possible map a portion of a client address space into a portion of a specific file. This permits construction of files that are larger than the maximum address space. The support provided also permits files, especially code files, to be shared between system components.

Client Synchronization Mechanisms

Within the kernel, semaphores and events permit control of access to kernel structures, such as control blocks. Similar access controls and concurrency control mechanisms must be provided to

clients. However, use of such features depends on the internal state of client objects, and client objects can be swapped from main memory, while kernel structures cannot. A set of higher level mechanisms use the kernel's synchronization mechanisms to implement facilities than enable clients to construct mechanisms that function similarly.

Primitive Transaction Coordination:

Transaction support for ArchOS is implemented in two portions. Higher level requirements, such as support for compound transactions, are implemented above the subkernel. The subkernel implements only support for the most elementary nested transactions. Support is provided for object operation invocation, transaction completion -- either commit or abort -- and orphan elimination. A new technique discussed in [Mckendry 85] permits bounds to be placed on the time until lock release as a result of aborted transactions. This feature will assist in scheduling transactions in real time.

6.2.3 ArchOS Base Kernel

The ArchOS base kernel uses facilities implemented by the Alpha sub-kernel to provide basic mechanisms for ArchOS facilities, thus creating a uniform arobject environment for system arobjects. Since the ArchOS kernel is not a traditional monolithic kernel, a set of kernel arbojects are running on the top of the base kernel and supporting necessary low-level functions for ArchOS facilities.

The major functionality of the base kernel can be summarized as follows:

- Creation and destruction of kernel arobjects and processes
- Communication between kernel arobjects
- Dispatching mechanism for the time-driven scheduler
- Address space management for time-driven virtual memory manager
- Low-level synchronization between processes and interrupt/trap

In other workds, the basic kernel is responsible to perform local resource management for creating and destructing kernel objects and for invoking its operations. Any remote or system-wide resource management decisions are decided based on coordination among kernel arobjects.

6.2.3.1 Kernel Arobjects and Processes

The ArchOS base kernel provides creation and destruction of a kernel arobject in its node. A kernel arobject is similar to a normal arobject except that it resides in a kernel address space and shares the address space among the other kernel arobjects. Similarly, all of the kernel processes share their address space even though each process logically belongs to a specific kernel arobject.

Binding of a kernel arbojject/process and a reference name is also managed in the same way. Unlike a client's reference name, a kernel arobject's reference name is treated as a "well-known" name representing a service class.

The following normal ArchOS primitives are used to manage kernel arobjects and processes.

```
arobject-id = CreateArojbect(arobj-name [, init-msg])
process-id = CreateProcess(process-name [, init-msg])
val = KillArobject(aid)
val = KillProcess(pid)

aid = SelfAid()
paid = ParentAid(aid)
pid = SelfPid()
ppid = ParentPid(pid-x)

val = BindArobjectName(aid, arobj-refname)
val = BindProcessName(pid, process-refname)
val = UnbindArobjectName(aid, arobj-refname)
val = UnbindProcessName(pid, process-refname)

aid = FindAid(arobj-refname)
pid = FindPid(process-refname)
aid-list = FindAllAid(arobj-refname)
pid-list = FindAllPid(process-refname)
```

AID arobject-id The unique identification of the instantiated arobject.

PID process-id The unique identification of the instantiated process.

AROBJ-NAME arobj-name
The name of arobject to be instantiated.

PROCESS-NAME process-name
The name of process to be instantiated.

MESSAGE *init-msg
A pointer to the initial message which contains initial parameters for the INITIAL process.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid

AID-LIST aid-list The list of corresponding aid's.

PID-LIST pid-list The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

It should be noted that the above primitives are identical to the ordinary ArchOS primitives clients can use, but only local creation and destruction of kernel arobjects and processes are supported in the ArchOS kernel. When an instance of a kernel arobject or process is created, an ordinary arobject or process descriptor will be created and managed uniformly at the kernel. (The description of the arobject/process descriptor is given in Section 4-1).

6.2.3.2 Kernel Communication Management

The ArchOS base kernel provides a local communication mechanism among kernel arobjects. A kernel arobject can invoke an operation at a single destination arobject or at multiple arobjects providing the same service name (i.e., sharing the same reference name).

---

trans-id = **Request**(arobj-id, opr, msg, reply-msg)
trans-id = **RequestSingle**(arobj-id, opr, msg)
trans-id = **RequestAll**(arobject-name, opr, msg)

pid = **GetReply**(trans-id, reply-msg)

(trans-id, requestor, opr) = **AcceptAny**(opr, msg)
(trans-id, opr) = **Accept**(requestor, opr, msg)

ptr-mds = **CheckMessageQ**(qtype, requestor, opr, req-trans-id)

trans-id = **Reply**(pid, req-trans-id, reply-msg)

TRANSACTION-ID trans-id
The transaction id of the transaction on whose behalf the request is being made.

AID arobj-id The unique id of the receiving arobject.

OPE-SELECTOR opr
The name of the operation to be performed.

MESSAGE *msg A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call-by-value semantics must be used)

REPLY-MSG *reply-msg
A pointer to the reply message.

MSG-DESCRIPTORS *prt-mds
Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype This indicates either "request-" or "reply-" message queue.

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr
The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

6.2.3.3 Policy Management

The policy management provides system functions to add, delete, and modify the policy definition module in ArchOS. Since the placement of the *policy definition module* is a major issue in terms of the system performance, ArchOS allows a client to specify the location by using a *policy definition descriptor*. The policy definition module consists of a *policy body* and a set of *policy attributes*. Both the policy body and attributes can be modified at runtime.

---

val = **SetPolicy**(policy-name, policy-def-desc)
val = **SetAttribute**(policy-name, attribute-name, attribute-value)

pdd = **AllocatePDD**()
val = **FreePDD**(pdd)

BOOLEAN val TRUE if the specified policy was set properly; otherwise FALSE.

PDD policy-def-desc
A pointer to the policy definition descriptior.

ATTRIBUTE-NAME attribute-name
The name of attribute to be set.

ATTRIBUTE-VALUE attribute-value
The actual value for the attribute.

The *SetPolicy* primitive links a user-defined policy definition module to the ArchOS base kernel. The *SetAttribute* primitive set a specific value(s) for its one of attribute. Since a policy definition descriptor is maintained in the base kernel, any access to the actual policy body or a value of its attribute can be easily made.

The *AllocatePDD* primitive allocates a policy definition descriptor in the base kernel and *FreePDD* releases the allocated descriptor.

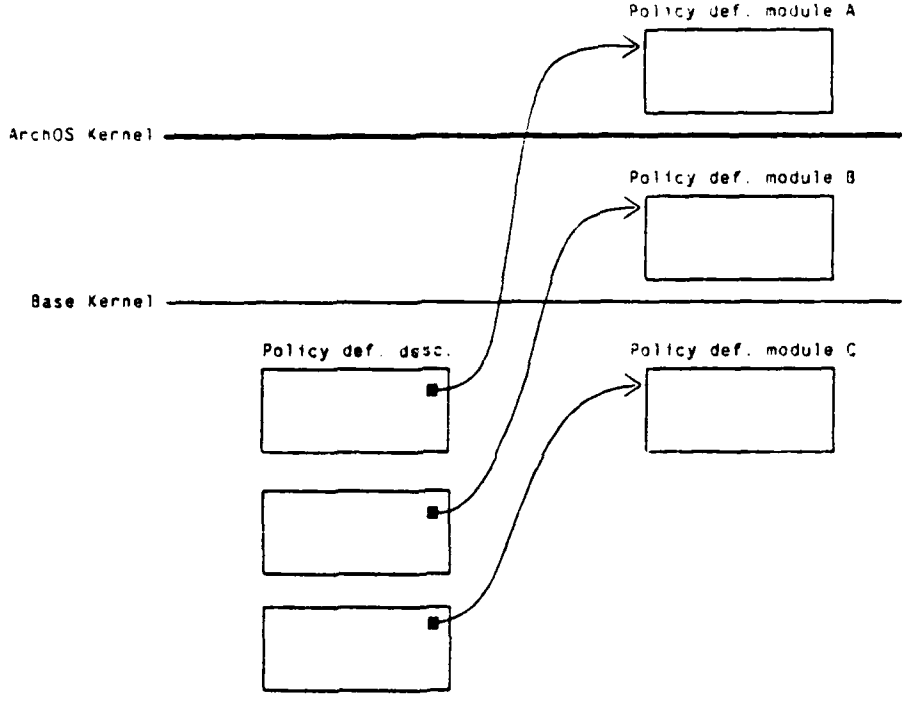


Figure 6-2: Policy Definition Module and PDD

6.2.3.4 Time-driven Scheduling Management

The interface between the scheduler and the remainder of the ArchOS kernel is a simple one in which the scheduler acts as a simple object providing operations and maintaining its own scheduling data base, including its scheduling queue. These operations, of course, will not be available to the ArchOS client, but will be used internally by ArchOS to generate scheduling requests whenever needed. The operations defined by the scheduler are as follows.

pid-list = **Schedule()**

pid-list = **RequestSwapList()**

SetScheduleInfo(pid)

Deschedule(pid)

PID-LIST pid-list The list of pids.

PID process-id The unique identification of the instantiated process.

The *Schedule()* primitive returns a list of the process ids (pids). The first pid indicates a process to be executed at this time and the following pids are candidates for the following scheduling point. No parameters are passed, and the scheduler makes its decision directly from the information within its data base.

The *RequestSwapList()* primitive returns a list of the pids which indicates candidates for the following swapping decisions at the virtual memory manager level.

The *SetScheduleInfo* primitive enters the necessary scheduling information for the specified process to the scheduler. The scheduler places the process pid and all its scheduling parameters into its database and prepares for making the scheduling decision required when the next **Schedule** operation is performed. This decision-making process operates continuously, concurrently with the application processing, preparing its next scheduling decision.

The *Deschedule* primitive removes a process pid from its queue, updating its current scheduling database to prepare for the next *Schedule* operation.

The Scheduler will use the interrupt mechanism in the processor running the ArchOS kernel to invoke other kernel mechanisms when it makes relocation decisions or must make decisions regarding process scheduling with respect to other nodes. In addition, the interrupt mechanism will be used to interrupt the kernel and application processing when sufficient time has elapsed that the current scheduling decision must be reconsidered.

6.2.3.5 Address Space Management

The base kernel provides a number of primitive, though powerful mechanisms for constructing, manipulating, and accessing the definitions of the virtual address spaces of processes. A process' virtual address space (or simply *address space*) is illustrated in Figure 6-3. An address space consists of four non-overlapping *regions*:

1. **Kernel Region:**
   The Kernel Region is identical for all processes, and is only accessible when executing in kernel mode. It contains all of the kernel code and data structures. The kernel is shared by all processes and (at least in the initial version of ArchOS) will be non-pageable.

2. **Private Region:**
   The Private Region is unique for each process, and it consists of five *segments*: Kernel Stack Segment, User Stack Segment, User Heap Segment, User Data Segment, and User Text Segment. Each of these segments will be discussed below.

3. **Shared Region:**
   The Shared Region is unique for each arobject, but shared by all processes within an arobject. This region contains all of the arobject's private abstract data type instances, along with the code for accessing and manipulating them. It consists of a variable number of segments, one (or more) for each instance of a shared abstract data type, plus a Shared Text Segment and Shared Headers Segment. Each of these different types of segments will be discussed below.

4. **Kernel Interface Region:**
   The Kernel Interface Region is quite small, and primarily contains the code and data areas needed for switching between user and kernel modes. This region is shared by all processes and is non-pageable.

The Kernel Region and Kernel Interface Region are of fixed size, and the virtual to physical address maps for these regions, as well as their protection attributes, do not vary from address space to address space (process to process), or during the lifetime of an address space. As a result, these two regions are constructed automatically whenever a new address space is created, and never altered thereafter. The remaining portion of an address space is comprised of the Private Region and Shared Region. The relative sizes of these two regions can vary from arobject to arobject. However, within an arobject, all processes share the same Shared Region, and hence all processes within a single arobject will have identically sized Private and Shared Regions.

The Private Region and Shared Region each consist of a number of (non-overlapping) segments, of varying sizes and types. The different types of segments have corresponding protection attributes. Some segments are required for each address space, while others are optional. Some segments can be expanded (assuming they have space to grow), while others have a fixed size once allocated. The relative locations of the various segments are somewhat constrained, and as a result, the order in

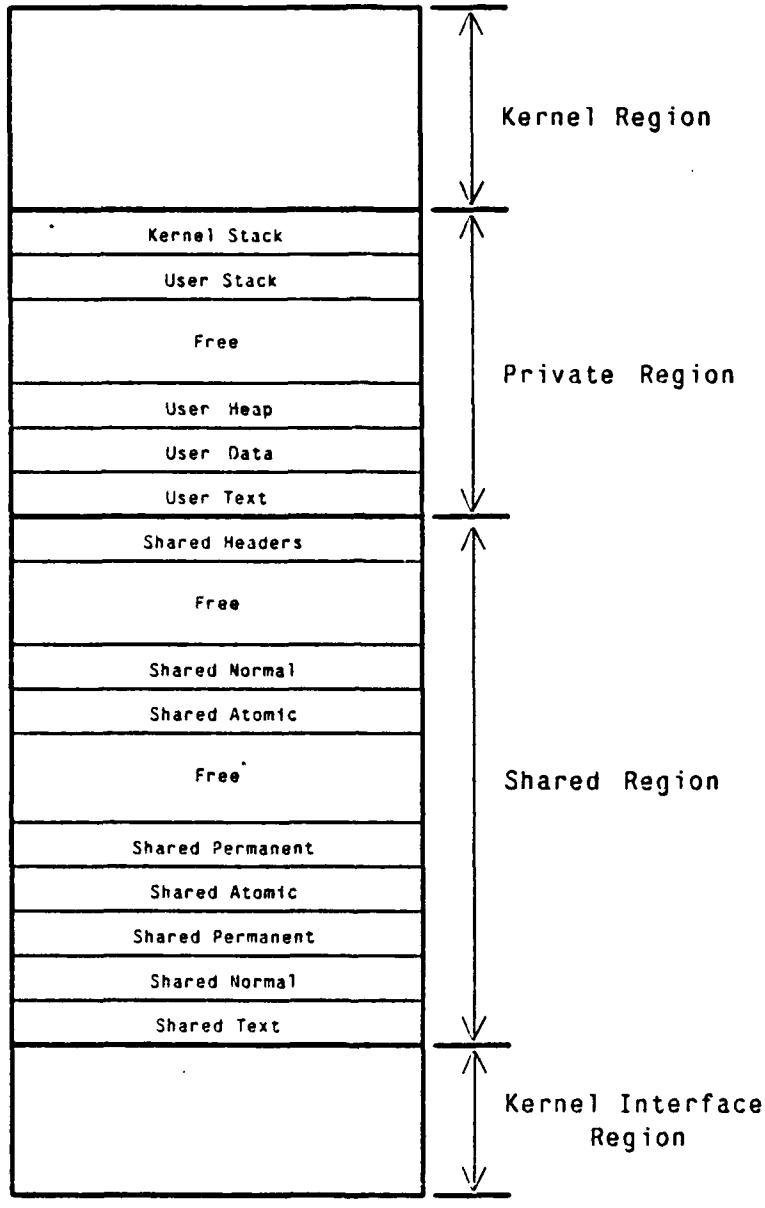


Figure 6-3: Virtual Address Space (One per Process)

which the segments can be (dynamically) allocated is similarly constrained. Each allocated segment has an associated page set, which is used as the paging area for that segment (see Section 6.7 for a description of page sets). The type of page set associated with a segment (*temporary*, *permanent*, or *atomic*) depends upon the type of the segment.

Private Region Segments

The Private Region contains five distinct segments, whose relative locations must be as shown in Figure 6-3:

Kernel Stack Segment:

The Kernel Stack Segment has a fixed size, which is the same for all address spaces, and is always located at the top of the Private Region. This stack is only accessible while in kernel mode, and it is "substituted" for the User Stack (see below) as part of the operation of switching to the kernel. Since the kernel is not pageable, the Kernel Stack Segment is also not pageable. Like the Kernel Region and the Kernel Interface Region, the Kernel Stack Segment is constructed automatically whenever a new address space is created, and never altered thereafter.

User Stack Segment:

The User Stack Segment is a required segment, and is located immediately below the Kernel Stack Segment. It can vary in size from address space to address space, and during the lifetime of an address space. The User Stack Segment is the only segment in the Private Region which grows downward. All other segments are either fixed size or grow upward to include larger virtual addresses. The User Stack Segment contains the subroutine stack (including parameters and local variables) while a process is executing in user mode. Hence, this segment must be both readable and writable from user mode. The User Stack Segment is pageable and should be associated with a *temporary* page set, which is to be used as the paging area.

User Text Segment:

The User Text Segment is also a required segment, and it is located at the bottom of the Private Region. It contains the code to be executed by the process, and hence its protection is set to allow only execute access while in user mode. The User Text Segment can vary in size from address space to address space, but once allocated it never changes.[4] This segment is pageable and should be associated with the *permanent* page set which contains the code for the process. Note that since the User Text Segment is not writeable, only "page-ins" (and no "page-outs") will ever be required. Furthermore, in order to reduce primary memory requirements, the Time-Driven Virtual Memory Subsystem (see Section 6.10) will arrange for User Text pages to be shared among all processes executing the same code on a single node.

User Data Segment:

[4] Due to limitations in the language compilers and linkers, the User Text Segment may contain the code for the entire project, rather than just the code required by a single process. In that case the User Text Segments for all processes within a single project will have the same size.

The User Data Segment is not strictly required, but it will almost always be present. It is located immediately above the User Text Segment, and should be allocated after the User Text Segment has been specified. The User Data Segment contains the "global", initialized variables that are used by (and private to) the process. It is both readable and writable while in user mode. The User Data Segment can vary in size from address space to address space, but once allocated it never changes.[5] This segment is pageable and should be associated with the *permanent* page set which contains the initialized data for the process. However, only (initial) page-ins will ever be performed using this page set, so as not to destroy the definitions of the *initial* values. All page-outs and subsequent page-ins will be to the same *temporary* page set used as the paging area for the User Stack Segment.[6]

User Heap Segment:

The User Heap Segment is not strictly required, but it too will almost always be present. It is located immediately above the User Data Segment, and should be allocated after the User Text Segment and User Data Segment have been specified. The User Heap Segment contains the dynamically allocated variables that are used by (and private to) the process. It is both readable and writable while in user mode. The User Heap Segment can vary in size from address space to address space, and during the lifetime of an address space. Note that since the User Heap Segment grows upward while the User Stack Segment grows downward, expansion of these two segments cause additional space to be allocated from opposite ends of the (single) free area within the Private Region. The User Heap Segment is pageable and should be associated with the same *temporary* page set which is used as the paging area for the User Stack Segment.[7]

Shared Region Segments

The Shared Region contains five distinct types of segments, although there can be multiple segments of a particular type. Also, the placement of those segments within the Shared Region is somewhat more flexible than the placement of the segments within the Private Region. The general structure of the Shared Region is illustrated in Figure 6-3. Note that the entire Shared Region is optional, and will only be present if the arobject definition includes one or more private abstract data

---

[5] As in the case of User Text Segments, limitations in the language compilers and linkers may cause the User Data Segments for all processes within a single arobject to have the same size.

[6] Multiple segments can use the same page set for paging purposes, by mapping the virtual page addresses within the various segments directly onto the same page numbers within the page set, i.e. virtual page *x* gets mapped into page number *x*. Since segments do not overlap, the corresponding paging areas will not overlap either. See Section 6.7 for more details on the use of the page set facilities.

[7] If desired, a separate (temporary) page set could be used, allowing paging to different disks, or even different nodes.

type definitions. Also note that the Shared Region is identical for all of the processes (address spaces) which belong to a single arobject, and are resident on a particular node. As a result, the Shared Region should only be modified in a single address space on a given node, in order to have all of the related address spaces on that node updated simultaneously.

The types of segments that can appear in the Shared Region are the following:

Shared Text Segment:

There is a single Shared Text Segment, and it must be present whenever the Shared Region exists. It is located at the bottom of the region, and contains the code which defines the permitted operations on the arobject's private abstract data types. The Shared Text Segment permits only execute access while in user mode. However, it is assumed that this code will always be entered "indirectly", by first determining which abstract data type instance is to be operated upon, and then directly invoking the requested operation with the specified instance as a parameter. Furthermore, it is assumed that operations will only be directly invoked on instances which are located on the local node.[8] The Shared Text Segment can vary in size from address space to address space, but for a given arobject, once it has been allocated it never changes. This segment is pageable and should be associated with the *permanent* page set which contains the code for the operations defined on the arobject's private abstract data types. Note that since the Shared Text Segment is not writeable, only page-ins (and no page-outs) will ever be required. Furthermore, in order to reduce primary memory requirements, the Time-Driven Virtual Memory Subsystem will arrange for Shared Text pages to be shared among all processes executing the same code on a single node.

Shared Headers Segment:

There is a single Shared Headers Segment, and it too must be present whenever the Shared Region exists. It is located at the top of the region, and contains the "header" information describing all instances of private abstract data types that have been created within the arobject. This header information includes the node on which each instance is located, and the address(es) of the Shared Normal Segment, Shared Permanent Segment, and/or Shared Atomic Segment associated with each instance. The header information is used whenever operations are to be invoked on specified abstract data type instances, and hence the Shared Headers Segment allows read-only access while

---

[8] Abstract data type instances can be distributed throughout the nodes of the computer system, although a single instance will always be completely contained within a single node. The indirect invocation of an operation on an abstract data type instance involves first using the information in the Shared Headers Segment to determine which node contains the instance in question. If the instance is on the local node, the operation can be invoked directly. However, if the instance is on a remote node, a form of remote procedure call (RPC) is used in order to invoke the operation on that remote node. See Section RPCSEC for more details on the use of the RPC facility to implement distributed private abstract data types.

in user mode. The Shared Headers Segment can vary in size from address space to address space, and during the lifetime of an address space. However, within a single arobject, all address spaces will have identical Shared Headers Segments. The Shared Headers Segment is the only segment in the Shared Region which grows downward. All other segments are either fixed size or grow upward to include larger virtual addresses. The Shared Headers Segment is pageable and should be associated with a *temporary* page set, which is to be used as the paging area. Note, however, that the Shared Headers pages are shared among all of the processes of a single arobject, which are executing on a particular node.

Shared Normal Segments:

There can be multiple Shared Normal Segments within the Shared Region, one for each abstract data type instance which contains "normal" shared data. These segments can be located almost anywhere within the Shared Region, above the Shared Text Segment and below the Shared Headers Segment. However, successive segments will normally be allocated immediately above the Shared Text Segment and any other existing Shared Normal, Shared Permanent, and Shared Atomic Segments. The lifetime of a Shared Normal Segment may be shorter than the lifetime of the address space, since abstract data type instances can be created and destroyed dynamically. Shared Normal Segments permit both read and write access while in user mode, since these segments contain the normal shared variables which define the current states of the abstract data type instances. In addition to being dynamically created and destroyed, Shared Normal Segments can be expanded (grown), to support the dynamic allocation of shared normal variables. Note that since these segments grow upward while the Shared Headers Segment grows downward, the free area near the top of the Shared Region (immediately below the Shared Headers Segment) will be allocated from opposite ends, reducing the chances of conflict. Shared Normal Segments are pageable and should be associated with *temporary* page sets, which are to be used as the paging areas.[9] Note that Shared Normal pages, like Shared Headers pages, are shared among all of the processes of a single arobject, which are executing on a particular node.

Shared Permanent Segments:

Shared Permanent Segments are almost identical to Shared Normal Segments, except that they contain the "permanent" shared variables which define the current states of the abstract data type instances. Also, each Shared Permanent Segment is associated with a *permanent* page set, which is

---

[9] It is possible to use a single temporary page set as the paging area for all Shared Normal Segments, as well as the Shared Headers Segment. However, the use of separate page sets allows the individual paging areas to be located on the same nodes as their corresponding abstract data type instances, increasing efficiency. In addition, it makes it easier to "free" Shared Normal Segments, and to move them around within the Shared Region. (This latter operation may be required in order to avoid conflicts when expanding existing segments.)

used as both its paging and permanent storage area.[10]

Shared Atomic Segments:

Shared Atomic Segments are identical to Shared Normal and Shared Permanent Segments in most respects, except that they contain the "atomic" shared variables which define the current states of the abstract data type instances. Because of this, each Shared Atomic Segment is associated with an *atomic* page set, which is used as both its paging and atomic (permanent) storage area.[11] In order to determine which portions (variables) within Shared Atomic pages have been modified by various transactions, direct writing to the Shared Atomic Segments is not permitted while in user mode (these segments are read-only in user mode). Instead, the special *AtomicCopy* primitive must be used in order to modify any atomic variables. As each modification is made to a Shared Atomic page, the *AtomicCopy* primitive also records the modification in the associated atomic page set. This permits the modification to later be committed or aborted under control of the Transaction Management Subsystem. Note that since each modification is immediately written to the atomic page set, there will never be any need for the Time-Driven Virtual Memory Subsystem to page-out Shared Atomic pages. Also, page-in operations only require the Shared Atomic page to be read from the atomic page set, since the Atomic Page Set Manager will ensure that the page, as read, will reflect all outstanding (uncommitted) modifications.[12] For more details on the handling of atomic objects and transactions, see Section TMSEC.

Address Space Management Primitives

The base kernel provides primitives for creating and destroying address spaces, for allocating, freeing, growing, and moving segments within an address space, and for accessing and modifying the information associated with each virtual page within an address space (mapping, flags, and time of

---

[10] Again, it is possible to use a single permanent page set to hold all of the Shared Permanent Segments, but the same considerations as for Shared Normal Segments make the use of separate page sets a bit more attractive.

[11] In this case too it is possible, though slightly less attractive, to use a single atomic page set to hold all of the Shared Atomic Segments.

[12] The consistency of the atomic data values accessed and modified by a transaction is not guaranteed by these facilities, unless the correct locking protocols are followed. It is assumed that the appropriate read and write locks are obtained for each atomic data item that is to be accessed or modified, respectively. Also note that transaction aborts require notifying the Atomic Page Set Manager of the abort, and also undoing all of the aborted modifications in all of the affected Shared Atomic pages. This latter operation is best accomplished by simply flagging all of the affected pages as "invalid", so that the next attempt to access them will result in a page-in operation. Since the modifications associated with the aborted transaction have been removed from the atomic page set, the page-in operation will read the properly modified contents of the page.

last access).[13] The actual address space management primitives provided by the base kernel are the following:

```
val = ASCreate(asid [, nvp-shared])
val = ASDestroy(asid)
val = ASActivate(asid)

vpa = ASAllocate(asid, seg-type, nvp, gpsid [, desired-vpa])
val = ASFree(asid, vpa)
val = ASExpand(asid, vpa, nvp)
val = ASMove(asid, vpa, desired-vpa)

ste = ASGetSTE(asid, vpa)
ste = ASNextSTE(asid, vpa)

val = ASSetMap(asid, vpa, ppa [, nvp])
ppa = ASGetMap(asid, vpa)
val = ASSetFlags(asid, vpa, flags [, nvp])
flags = ASGetFlags(asid, vpa)
val = ASSetTime(asid, vpa, time [, nvp])
time = ASGetTime(asid, vpa)
val = ASSetPTE(asid, vpa, pte [, nvp])
pte = ASGetPTE(asid, vpa)

(gpsid, pnum) = ASGetGPSID(asid, vpa)
(asid, vpa, ppa) = ASFindShared(gpsid [, pnum])
```

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

VIRT-PAGE-ADDRESS vpa, desired-vpa
: A virtual page address within an address space.

SEG-TABLE-ENTRY ste
: A descriptor for a segment within an address space. It includes the segment type (*seg-type*), location (*vpa*), size (*nvp*), and associated page set (*gpsid*).

PHYS-PAGE-ADDRESS ppa
: The physical page address in primary memory, to which a virtual page address is mapped.

PTE-FLAGS flags — The set of (BOOLEAN) flags associated with a particular page in an address space: USED, MODIFIED, VALID, EXISTS, and COPIED.

---

[13] No facilities are provided for accessing or modifying the protection codes governing access to the various parts of an address space. These protection codes are set automatically whenever the address space is created, and whenever new segments are allocated. There should never be any need to modify them explicitly.

VIRT-TIME time — The (approximate) virtual (CPU) time at which a particular page in an address space was last accessed.

PAGE-TABLE-ENTRY pte — A descriptor for a particular page in an address space. It includes the corresponding physical page address (*ppa*), *flags*, and last access time (*time*).

GPSID gpsid — Global Page Set ID, which identifies the page set associated with a segment. It includes the ID of the logical disk containing the page set, the page set type (TEMPORARY, PERMANENT, or ATOMIC), and the unique ID of this page set within the logical disk.

INT pnum — A page number within a page set, corresponding to a virtual page within an address space segment.

ASID asid — Address Space ID, which identifies the address space to be operated upon. The corresponding process ID can be easily obtained from an ASID, and vice versa.

INT nvp-shared — The number of virtual pages (size) of the Shared Region.

SEGMENT-TYPE seg-type — The type of the address space segment to be allocated: USER-STACK, USER-TEXT, USER-DATA, USER-HEAP, SHARED-TEXT, SHARED-HEADERS, SHARED-NORMAL, SHARED-PERMANENT, or SHARED-ATOMIC.

INT nvp — The number of virtual pages involved in the operation.

On Error: — Error conditions are indicated by the use of special return values. The details concerning the precise nature of an error condition are provided in the Kernel Error Block.

The *ASCreate* primitive creates a new address space, corresponding to a newly created process.[14] The address space ID (*asid*), which is closely related to the process ID (allowing easy translation back and forth), must be specified as part of the operation. This *asid* will be used to identify the address space in all subsequent operations. The size of the Shared Region (*nvp-shared*) must also be specified if this is the first address space belonging to the corresponding arobject to be created on this node. Otherwise it is optional.[15] Since there is only a fixed, maximum amount of buffer space

---

[14] It can also be used to *recreate* an address space for a process which had been completely "swapped out", but will soon be required to run again. See Section 6.10 for more details about process swapping.

[15] Since the sizes of the Kernel and Kernel Interface Regions are fixed, specifying the size of the Shared Region will also determine the size of the Private Region (there are no other regions in an address space). Also, if another address space from the same arobject already exists on this node, the size of the Shared Region is already known (all address spaces from the same arobject have identical Shared Regions). The arobject ID can be easily determined from the *asid* (or its associated process ID).

available for storing address space definitions, the *ASCreate* primitive can fail if too many address spaces have already been defined.[16] The *ASDestroy* primitive can be used to destroy a previously defined address space.

ASActivate is used when switching processes. It makes the specified address space (*asid*) the currently active one, i.e. it switches the processor to that address space. Note that since the Kernel Region is identical for all address spaces, only the Private and Shared Regions are actually affected by this switch. The processor continues to execute the same code within the kernel as it was prior to switching address spaces. Depending upon the architecture of the system's memory management hardware, activating an address space may require the explicit loading of many registers within the Memory Management Unit (MMU). The management of these MMU registers is solely the responsibility of the address space management routines, especially *ASActivate*. Of course, MMU register management is simplified considerably if the MMU itself handles the loading of its mapping registers, in the manner of a cache.

ASAllocate allocates a new segment within an existing address space. The type of segment (*seg-type*) determines many of its characteristics. In most cases only one segment of a particular type is permitted within an address space. However, multiple SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments are allowed. At the time a segment is allocated, its initial size (*nvp*) and associated page set (*gpsid*) must be specified.[17] In most cases the location of a new segment is either fixed or can be determined automatically, assuming the various segment types are allocated in the proper order.[18] However, the exact location for a new segment can be specified (*desired-vpa*), whenever necessary.[19] *ASAllocate* returns the location (*vpa*) of the newly allocated segment. For all segments except USER-STACK and SHARED-HEADERS, this is the location of the first (lowest address) page in the segment. For USER-STACK and SHARED-HEADERS, the returned location is the last page in the segment. The returned *vpa* will be used to identify the segment in subsequent operations. *ASAllocate* will fail, returning BAD-VPA, if any conflicts are detected, such as attempting to allocate an already allocated segment type, or overlapping an existing segment.

---

[16] One remedy for this is to swap out and then destroy one of the already existing address spaces.

[17] This implies that the associated page set must already exist.

[18] The only restrictions on the ordering of segment allocations are that USER-HEAP must follow USER-DATA, which in turn must follow USER-TEXT, and SHARED-NORMAL, SHARED-PERMANENT, or SHARED-ATOMIC segments must be allocated after the SHARED-TEXT segment.

[19] This should only be necessary when constructing the Shared Region on a new node, so that it matches the Shared Region for an object that already exists on other nodes.

ASFree deletes an entire segment, indicated by *vpa*, within the specified address space (*asid*). Only SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments can be freed.[20] *ASExpand* increases the size of an existing segment by the specified number of pages (*nvp*). It will fail if there is insufficient space for the segment to grow by the amount indicated. *ASMove* changes the location of an existing segment from *vpa* to *desired-vpa*. Only SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments can be moved.[21] *ASMove* will fail if the new location for the segment would overlap another existing segment.

ASGetSTE returns a descriptor for the segment which contains the specified virtual page (*vpa*). This descriptor indicates the segment's type (*seg-type*), location (*vpa*), size (*nvp*), and associated page set (*gpsid*). BAD-STE is returned if the specified page is not within one of the existing Private Region or Shared Region segments.[22] *ASNextSTE* is similar to ASGetSTE, except that it returns a descriptor for the next (higher address) segment which follows, but does not contain, the specified virtual page. Specifying *vpa = 0* will cause the descriptor for the first (lowest address) segment in the Shared Region (SHARED-TEXT) to be returned, or the descriptor for USER-TEXT to be returned if there is no Shared Region. BAD-STE will be returned if the specified page is within or beyond the last segment of the Private Region (USER-STACK). *ASNextSTE* is useful for scanning through all of the segments (and pages) which constitute an address space.

ASSetMap sets the virtual to physical mapping for virtual page *vpa*, to physical page *ppa*. Optionally, a range of *nvp* virtual pages, beginning with *vpa*, can be mapped to contiguous physical pages, beginning with *ppa*. *ASGetMap* returns the physical page address (*ppa*) to which the specified virtual page (*vpa*) is mapped. BAD-PPA is returned if the mapping has not been previously defined using *ASSetMap* (or *ASSetPTE*). *ASSetFlags* sets all of the (BOOLEAN) flags associated with the specified virtual page (*vpa*). Optionally, the flags for a range of *nvp* virtual pages, beginning with *vpa*, can all be set to the same values (*flags*). The available flags are:

- USED: The virtual page has been accessed. This flag helps determine which pages belong to the working set of a process (See Section 6.10).
- MODIFIED: The virtual page has been modified. This flag indicates that the page must be written to the associated page set before the physical page frame can be reused.

---

[20] This happens as a result of destroying abstract data type instances.

[21] Moving of segments can be a useful way to recover from *ASExpand* failures.

[22] For our purposes here, the Kernel Stack Segment is not considered a part of the Private Region. Only the "USER" segments are

- VALID: The physical page address to which the virtual page is mapped is valid, i.e. the page is in primary memory.

- EXISTS: The virtual page exists in the associated page set, i.e. the page has been written some time in the past. This flag helps avoid page-in operations when a newly allocated virtual page is first accessed.[23]

- COPIED: The virtual page (within the User Data Segment) has been "copied" to the temporary page set, i.e. the temporary page set contains a newer version of the page than the permanent (initial data) page set. This flag only applies to the User Data Segment, and it is used to indicate which page set is to be used when paging-in the virtual page.[24]

ASGetFlags returns the set of flags associated with the specified virtual page (*vpa*).

ASSetTime sets the time of last access for the specified virtual page (*vpa*) to *time*. The *time* value is in virtual (CPU) time units. Optionally, the last access time for a range of *nvp* virtual pages, beginning with *vpa*, can all be set to the same value of *time*. *ASGetTime* returns the last access time for the specified virtual page. NEVER is returned if the page has never been accessed. *ASSetPTE* is equivalent to *ASSetMap*, *ASSetFlags*, and *ASSetTime* combined. It sets all three items of the virtual page descriptor(s) (*ppa*, *flags*, and *time*) to the specified values (*pte*). Similarly, *ASGetPTE* is equivalent to *ASGetMap*, *ASGetFlags*, and *ASGetTime* combined. *ASSetPTE* and *ASGetPTE* are provided as a convenience, for use when entire page descriptors must be modified or retrieved.

ASGetGPSID returns the global page set ID (*gpsid*) and page number (*pnum*) associated with the specified virtual page. This is the page set ID and page number to be used when paging-in or paging-out this virtual page.[25] BAD-GPSID and BAD-PNUM are returned if the specified page is not contained within any of the existing segments in the Private Region or Shared Region. *ASFindShared* searches for the specified page number (*pnum*) from the given page set (*gpsid*), to see if it is already resident in primary memory, and in active use within one of the existing address spaces.[26] If *pnum* is not specified, the search is for any active page from the given page set. If the search is successful,

---

[23] It isn't incorrect to page-in a nonexistent page. It would simply be read as all zeros (see Section 6.7). However, the EXISTS flag helps avoid the overhead of the (unnecessary) page-in operation. Whether a nonexistent page is initialized to zero on first access or simply left undefined depends upon the type of segment it belongs to. User Stack Segment pages can be left undefined, but pages in most other segments should be initialized to zero.

[24] The COPIED flag is essentially another name for the EXISTS flag, as it applies to the User Data Segment. Each User Data Segment page is known to exist in the permanent (initial data) page set. The only question is whether a newer version also exists in the temporary page set.

[25] *ASGetGPSID* takes the COPIED flag into account when determining the page set to be used for pages in the User Data Segment.

[26] This involves searching through the existing address spaces for any segments having *apsid* as the corresponding page set. The virtual page descriptor for the page corresponding to *pnum* is then checked to see if it is VALID.

the address space ID (*asid*) and virtual page address (*vpa*) of the first encountered matching entry is returned, along with the corresponding physical page address (*ppa*). Otherwise BAD-ASID, BAD-VPA, and BAD-PPA are returned. *ASFindShared* aids in the handling of "shareable" segments, such as the User Text Segments and all of the Shared Region segments. In particular, it can help determine if page-in or page-out operations are actually required.

Address Space Management Data Structures

The information describing the existing address spaces is contained within kernel data structures called *address space descriptors*. These descriptors are linked together in groups according to the arobjects to which the address spaces belong, as shown in Figure 6-4. This aids in the handling of the Shared Regions of address spaces, especially their construction and modification, since it allows all of the related address spaces to be updated "simultaneously".

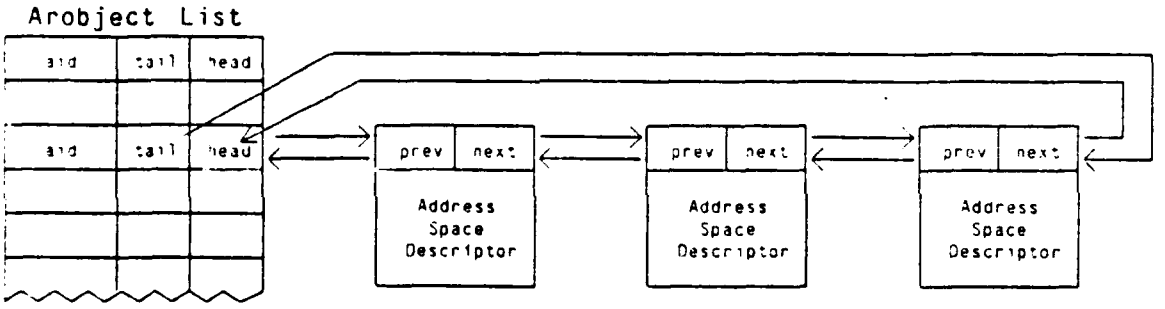


Figure 6-4: Arobject Address Space Lists

The structure of each address space descriptor is illustrated in Figure 6-5. It basically consists of a list of *Segment Descriptors*, which describe each of the segments contained within the address space. Note that since there can be varying numbers of Shared Normal, Shared Permanent, and Shared Atomic Segments, each of these types has its own (sub)list of segment descriptors. Any segment type which is not present in an address space would be indicated by a page count of zero (*nvp = 0*). Each segment descriptor includes a pointer (*pta*) to the *page table*, which describes the state of the individual pages of the segment. Note that each address space within an arobject will actually have its own set of page tables, even for the Shared Region. This is because the Shared Region can be accessed and used in very different ways by the different processes of an arobject, and it is important to determine the virtual memory "working sets" on a per process (per address space) basis (see Section 6.10).

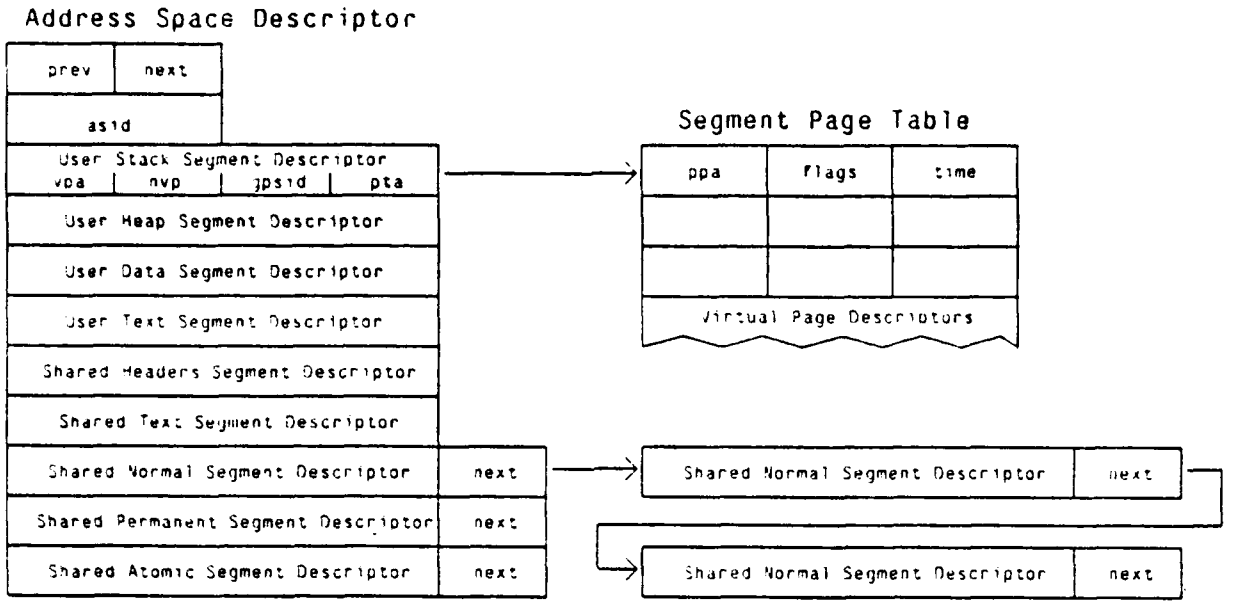


Figure 6-5: Address Space Descriptor

The other main data structure used in the management of address spaces is the Shared Page Set List, illustrated in Figure 6-6. The sole purpose of this structure is to improve the efficiency of the *ASFindShared* primitive.[28] Before paging-in or paging-out a potentially shared page, i.e. one from the User Text Segment or any segment within the Shared Region, the Time-Driven Virtual Memory Subsystem must first check (using *ASFindShared*) to see if the required page is already in main memory and in use by some other process. If so, the paging operation can be avoided. Given the global page set ID (*gpsid*) for the potential paging operation, *ASFindShared* will look for that *gpsid* in the Shared Page Set List, and then check each associated Segment Page Table to see if the page in question is ever listed as "VALID". Thus, the Shared Page Set List contains an entry for every page set corresponding to a User Text Segment or Shared Region Segment, in any existing address space. Associated with each entry is a list of all the segments (indicated by their address space IDs, virtual page addresses, and page table pointers), which share the use of that page set.

[28] Since the Shared Page Set List could be constructed solely from the contents of the Address Space Descriptors, its use is not strictly required. However, it greatly increases the efficiency of searching for shared pages.

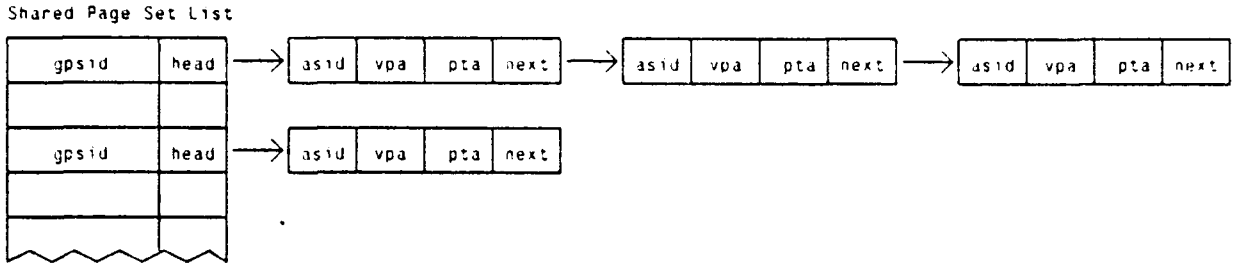


Figure 6-6: Shared Page Set List

6.2.3.6 Synchronization Management

The low-level synchronization mechanisms for arobjects and for basic I/O handling functions are provided at the base kernel. These primitives are designed as a local synchronization mechanism, so remote invocation is not supported at this level. All higher-level synchronization must be performed by using the communication primitives.

The following primitives are provided at the base kernel:

evtcnt = **Sigsend**(event-var)
evtcnt = **Sigrec**(event-var, timeout)
evtcnt = **Sigrecall**(event-var, timeout)
evtcnt = **Sigabort**(event-var, abort-code)

EVTCNT evtcnt The counter value of the specified event variable.

EVENT-VAR event-var
The event variable consists of a waiting queue of client processes and an event counter.

TIMEOUT timeout The timeout value should indicate the maximum execution time for this signal primitive including the waiting time.

ABORTCODE abort-code
An integer value which indicates an abort code.

The *Sigsend* and *Sigrec* primitives are basically similar to the V- and P-operations of an integer semaphore. However, the *Sigrec* primitive will be timed out if the corresponding signal (e.g., a hardware interrupt) is not generated. A *Sigrecall* primitive is similar to *Sigrec* and used for receiving

all stored event signals. A *Sigabort* primitive can be used to unblock the waiting process with an error condition.

6.3 Arobject/Process Management Subsystem

The Arobject/Process Management Subsystem is responsible for providing arobject/process management facilities in ArchOS. The subsystem consists of a Arobject/Process Manager and its worker processes on each node. The Arobject/Process Manager provides a system-wide facility in corporation with the base kernel. The workers are provided to perform actual work or decision making among cooperating Arobject/Process Managers in the system.

The Arobject/Process Manager primary responsible the following operations:

- Creation and destruction of an arobject and process
- Freezing and Unfreezing of an arobject's or process's activities
- Binding and unbinding of reference names for arobjects and processes
- Allocation and deallocation of private data objects
- Internal access mechanisms to fetch and store any data objects for an arobject/process.
- Recovery management for atomic arobjects

It should be noted that many functions can be invoked locally by the base kernel. This subsystem is necessary to provide the service for remote invocations.

6.3.1 Arobject/Process Management

An Arobject/Process Manager exists on each node and manages a fixed number of workers. Every worker can perform the following service functions:

- Coordinate with the other Arobject/Process Manager to perform the best assignment decision for creation of a new arobject/process instance.
- Propergate a new reference name to the other Arobject/Process Manager.

The basic components of the arobject/process subsystem is shown in Figure 6-7.

When a new instance of an arobject or process is created, a system-wide unique identifier called an *arobject id (aid)* or *process id (pid)* is created and is also guaranteed to be unique over the lifetime of the arobject or process. For a new instance of an arobject, an *arobject descriptor* is created and

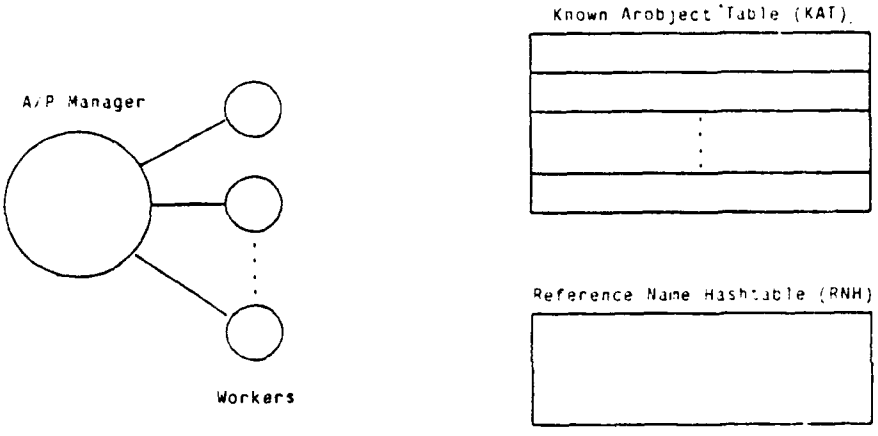


Figure 6-7: Components of the Arobject/Process Subsystem

registered in the *known arobject table* in the kernel. For a new process instance, a *process descriptor* is also allocated and linked to its arobject descriptor.

The arobject descriptor contains the following information to control its arobject components:

- Arobject id (aid):
  An aid is a fixed-length descriptor consisting of *current node id*, *birth node id*, and *local unique id*. Aid is used to identify a destination arobject where a request operation will be invoked, so the current node id and birth node id are included for reducing the searching process and migration process.

- Parent aid:
  Its parent's aid.

- Arobject status:
  Status of the arobject.

- Freeze/Unfreeze event variable:
  An event variable to control .

- A set of reference pointers to private object descriptors:
  A private object descriptor contains the data associated with the arobject's private object, namely "processes", "shared abstract data types", "statistics data object", "message queue", etc.

For instance, suppose that arobject A has two processes (INITIAL and P_1) and one shared private object (SO_1) and a new instance of A is created at node X. After the INITIAL process is started, it creates process P_1 at node Y. Then, the relationship among the major kernel objects are shown in Figure 6-8.

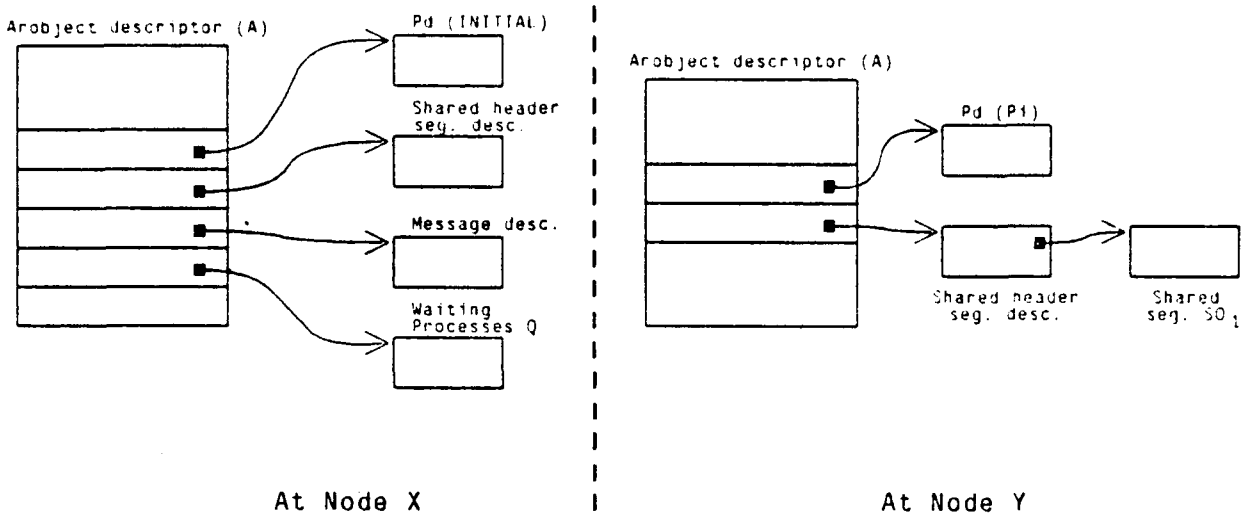


Figure 6-8: An example of Arobject Descriptors

At node X, an arobject descriptor is allocated for an instance of arobject A. A process descriptor for INITIAL and a shared object descriptor for the header segment and for SO_1 are linked to the arobject descriptor. At node Y, there are also an arobject descriptor for A and a process descriptor for P_1 which is linked to the arobject descriptor. Although there is no shared private object created at node Y, a shared object descriptor is also allocated for the header segment and linked to the arobject descriptor.

It should be noted that the message queue of arobject A is only allocated on node X. Thus, when P_1 attempts to accept a request message, a remote operation is invoked to fetch a message from the message queue at node X. If there is no acceptable request message in the queue, P_1 will be blocked and placed in the waiting processes' queue.

6.3.1.1 Arobject/Process Assignment Policy

When creation of a new instance of an arobject or process is requested at a non-specific node, the arobject/process manager selects the best node according to the current "arobject/process assignment policy".

The assignment policy, like other user definable policies, can be set by a system designer. In order to reduce system overhead, the policy definition module for the assignment policy is placed in Arobject/Process Manager.

Without creating a new policy definition module, the Arobject/Process Manager can provide the following assignment policies.

- First Fit (FF): The first A/P manager which replies the creation request will be selected.
- Random Fit (RF): A A/P manager from the random selection will be used.
- Best Fit (BF): One of the best matched A/P manager will be selected.
- Best Effort Fit(BEF):

6.3.1.2 Life cycle of Arobject/Process

The life cycle of an arobject depends upon whether the arobject is an atomic or nonatomic. When an instance of an arobject is instantiated, the arobject instance becomes *active*. If an arobject is atomic, it can be *inactivated* and remained on stable storage. On the other hand, if an arobject is nonatomic, it cannot be inactivated and it must be *dead* when it is killed. Atomic and nonatomic arobject can be *frozen* for monitoring or debugging purpose.

The life cycle of process is similar to the arobject's one. When an instance of a process is created, it becomes *ready* and runnable. Once the time-driven scheduler deciedes to run a process, it becomes *running* and the process may block due to I/O waiting or scheduler's preemption. Like arobject, a process will be *dead* when it is killed and it can be also *frozen*. The life time of a process instance depends on the life time of its arobject. When the arobject is killed, all of its processes will be also killed by the system. Thus, there will be no frozen processes left in its arobject.

The life cycle of an arobject and process is depicted in Figure 6-9.

6.3.1.3 ArchOS primitives

The following ArchOS primitives are supported for arobject/process management for a client.

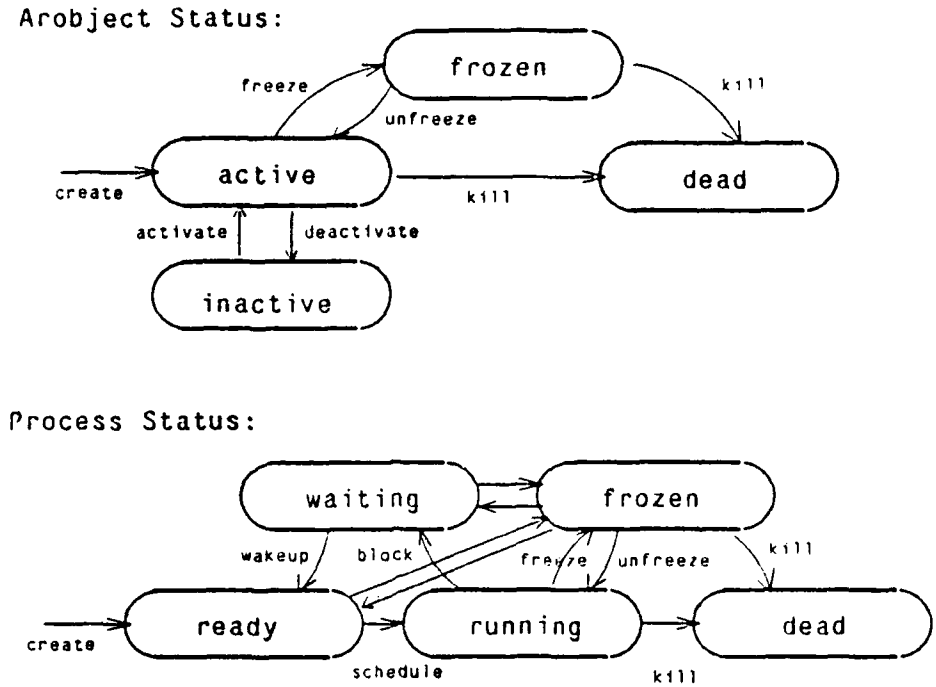


Figure 6-9: Life Cycle of an Arobject and Process

arobject-id = **CreateArojbect**(arobj-name [, init-msg] [, node-id])
process-id = **CreateProcess**(process-name [, init-msg] [, node-id])
val = **KillArobject**(aid)
val = **KillProcess**(pid)
val = **GlobalKillArobject**(arobj-id, kill-options)
val = **GlobalKillProcess**(pid, options)

aid = **SelfAid**()
pid = **SelfPid**()
paid = **ParentAid**(aid)
ppid = **ParentPid**(pid)

val = **SetErrorStack**(errblock, blocksize)
val = **FreezeArobject**(arobj-id, options)
val = **UnfreezeArobject**(arobj-id, options)
val = **FreezeProcess**(pid, options)
val = **UnfreezeProcess**(pid, options)

fval = **FetchArobjectStatus**(arobj id, dataobj-id, buffer, size)
sval = **StoreArobject**(arobj id, dataobj-id, buffer, size)
fval = **FetchProcessStatus**(pid, dataobj id, buffer, size)
sval = **StoreProcess**(pid, dataobj id, buffer, size)

BOOLEAN val TRUE if the primitive was executed properly; otherwise FALSE.

NODE-ID node-id The node id indicates the actual node which will be stopped.

AID arobj-id The unique arobject id of an arobject instance.

PID pid The process id of the target process.

GKILL-OP kill-options
The options indicate various control options. For example, it can indicate whether the caller stops every time after killing a single process or not.

FREEZE-OPT options
The options indicate various selectable flags such as a timeout freeze/unfreeze flag.

INT fval The actual number of bytes which were fetched.

INT sval The actual number of bytes which were stored.

DATAOBJ-ID dataobj-id The dataobj-id indicates the private object or system control status of the target arobject/process.

BUFFER *buffer A pointer to the buffer area for storing the returned data object value.

INT size The size indicates the buffer size in bytes.

A *CreateArobject* primitive creates a new instance of an arobject at an arbitrary node or a specified node. Similarly, a *CreateProcess* primitive creates a new instance of a process in the arobject. The selection of a node is made automatically by ArchOS unless overridden by the *Create* operation. Upon arobject creation, the arobject's INITIAL process is automatically dispatched. An optional set of parameters can be passed to the INITIAL process when the arobject is instantiated by using an initial message (i.e., "init-msg").

The *KillArobject* and *KillProcess* primitives remove a process and arobject instance respectively. An arobject may be killed only by one of its own processes (suicide allowed, no murder). In order to kill another arobject, the target arobject must have an appropriate operation defined within its specification so it can kill itself. A process can be killed only by a process which exists in the same arobject instance.

The *SelfAid* primitive returns the requestor's arobject id and the *ParentAid* primitive returns the

parent's arobject id of the specified arobject. The *SelfPid* primitive returns the process id (pid) of the requestor and the *ParentPid* primitive returns the parent's pid of the the specified process.

A *SetErrorBlock* primitive sets an error block in a process's address space. A user error block consists of a head pointer and a circular queue. The head pointer contains a pointer to an entry which contains the latest error information in the circular queue. After the execution of this primitive, a client can access the detailed error information from the specified error block. A *FreezeArobject* primitive stops the execution of an arobject (i.e., all of its processes), and a *FreezeProcess* primitives halts a specific process for inspection. An *UnfreezeArobject* and *UnfreezeProcess* primitive resumes a suspended arobject and process respectively. While a process is in a *frozen* state, many of the factors used for making scheduling decisions can be selectively ignored. For instance, a timeout value will be ignored by specifying a proper flag in the *Freeze* primitive.

A *Fetch* primitive inspects the status of a running or frozen arobject or process in terms of a set of frozen values of private data objects. The specific state of the arobject or process will be selected by a data object id. The state includes not only the status of private variables, but also includes process control information.

A *GlobalKill* primitive can destroy an arbitrary arobject or process in the system.

6.3.1.4 Creation and Destruction of Arobject/Process

A client can create a new instance of arobject at any node in the system by issuing the CreateArobject primitive. The first argument, *arobj-name*, contains three values: The first value contains a file name for the image of the shared region and the second value points to a file for the image of the private region of its INITIAL process. The last section contains a table of entry points for light-weight processes. Note that the light-weight process is available for kernel arobjects.

For instance, arobject name A may contains {"/usr/test/a.arobj", "/usr/test /p0.proc", nill}. When a client executes 'CreateArobject(A, msg, Y)", at first the base kernel determines whether the target node is local or remote. Since this is a remote invocation request, it looks up the A/P manager's *reference name hash-table* and determines the destination A/P manager's AID. Then, the client sets up a request message for the remote A/P manager and issues a proper invocation request.

When the remote Communication manager's NetIn worker receives the request packet, it placed in the target A/P manager's request queue. If one of its workers is already waiting on the incoming invocation request, the request message will be placed directly into the worker's message buffer.

Once the worker executes the CreateArobject primitive, the virtual address space for arobject A is created by reading "/usr/test/a.arobj" and "/usr/test p0.proc" files. A new arobject descriptor is also allocated and its AID is returned to the worker. Then, the worker returns the result message to its A/P manager and the A/P manager forward the result to the caller's A/P manager. Then, the original caller receives the result from the local A/P manager.

The sequence of interaction between two A/P manager is depicted in Figure 6-10.

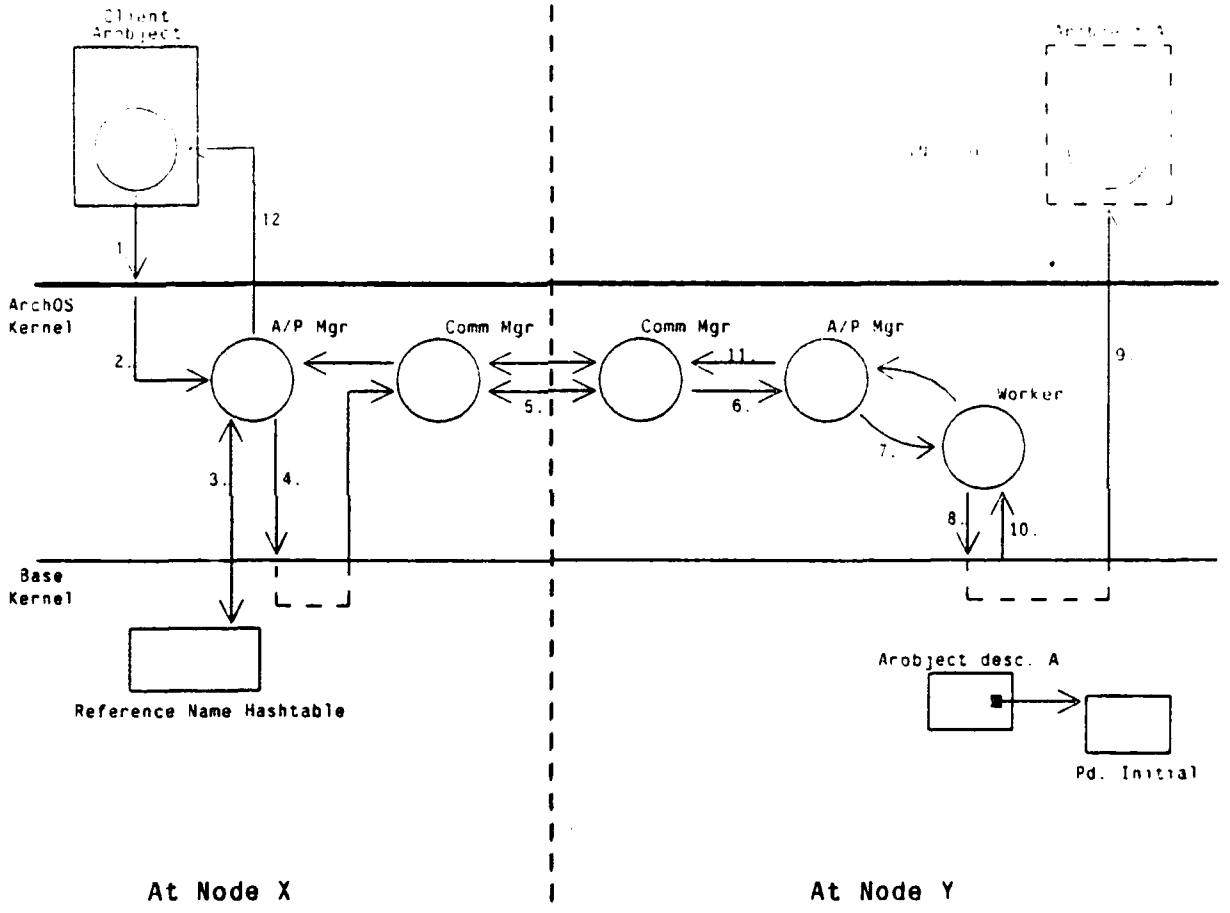


Figure 6-10: Creation and Destruction of an Arobject and Process

6.3.2 Name Management

An arobject/process manager maintains binding information in a hash table. When a reference name is bound to a caller, the caller's arobject/process manager registers its name first. Then, the manager's worker must propagate its entry across the system using one to many communication (i.e. using a *RequestAll* primitive). The lifetime of a binding is the same as the lifetime of an arobject or process instance.

To find one or all arobject/process instances from its reference name, the arobject/process manager searches its own hash table and if there is no entry there, then it inquires from the other managers.

The following ArchOS primitives are supported for name management for a client:

---

val = **BindArobjectName**(aid, arobj-refname)
val = **BindProcessName**(pid, process-refname)
val = **UnbindProcessName**(pid, process-refname)
val = **UnbindArobjectName**(aid, arobj-refname)

aid = **FindAid**(arobj-refname [, preference])
pid = **FindPid**(process-refname [, preference])
aid-list = **FindAllAid**(arobj-refname [, preference])
pid list = **FindAllPid**(process-refname [, preference])

AID aid — The arobject id.

PID pid — The process id.

AID-LIST aid-list — The list of corresponding aid's.

PID-LIST pid-list — The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

PREFERENCE preference
The preference can specify a search domain such as "INTERNAL", "EXTERNAL", "LOCAL", "REMOTE", "INTERNAL-LOCAL", "INTERNAL-REMOTE", "EXTERNAL-LOCAL", "EXTERNAL-REMOTE".

The *BindArobjectName* and *BindProcessName* primitives bind the requested instance of an arobject

or process to a reference name. This binding allows an arobject or a process to have more than one reference name, or a single reference name can be bound multiple arobject or process instances. To cancel the current binding, a process must use the appropriate *Unbind* primitive.

A *FindID* primitive returns the unique id (i.e., aid or pid) of the given arobject or process in a specific search domain. A search domain can be specified with respect to all of the internal arobjects, external arobjects, a local node, a remote node, or a reasonable combination of among four. If more than one instance uses the same reference name, the unique id of any one of them will be returned. A *FindAllID* primitive, on the other hand, returns all of the aid's and pid's which correspond to the given reference name.

6.3.3 Private Object Management

Private object management allows a client process to allocate and deallocate an instance of a private abstract data type at any node. An Arobject/Process manager maintains a list of private object descriptors under its arobject descriptor to keep the data associated with it.

Since an object type can be one of *Normal*, *Permanent*, and *Atomic*, the private object manager must coordinate with a page set manager (see Section PAGESET) to allocate a proper type of page set to create a new instantiation of the abstract data type.

If a remote allocation is requested, the creation of a new private abstract data type must be coordinated between the destination's Arobject/Process manager and the one with the INITIAL process. Since every process share all private segments and should have a uniform view of local and remote instances, the shared header segment must be updated by the Arobject/Process manager at the INITIAL process's node. When a proper update is done, updated part of the shared header segment is propagated to the other Arobject/Process manager.

---

object-ptr = **AllocateObject**(type-name, object-type, parameters, [, node-id])
val = **FreeObject**(object-ptr)
val = **FlushPermanent**(object-ptr, size)

OBJECT-PTR object-ptr
A pointer to the allocated private data object.

OBJECT-TYPE object-type
The object-type indicates the name of a private abstract data type.

BOOLEAN val TRUE if the object was released successful; otherwise FALSE.

NODE-ID node-id Node identification. An actual node may be designated, or a node selection

	criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).
INT size	The number of bytes which must be flushed into permanent storage.

An *AllocateObject* primitive allocates an instance of a private abstract data type at any node and a *FreeObject* primitive deallocates the specified instance. A *FlushPermanent* primitive blocks the caller until the specified data object is saved in non-volatile storage.

6.3.4 Recovery Management

The arobject process manager is responsible for restarting *atomic arobjects* which have at least one private atomic data object in the event of node failures. Since all private atomic objects are kept on a corresponding atomic page set, the arobject/process manager will coordinate with the page set subsystem to resume the crashed atomic arobject by recreating its initial process with the pre-crash image of the private atomic data objects and permanent data objects, if any.

It should be noted that it is still an application designer's responsibility to determine what recovery action must take place based on its atomic and permanent data objects.

6.4 Communication Subsystem

The communication subsystem provides intra- and inter-node message communication mechanisms to support a system-wide, location- independent operation invocation for cooperating arobjects. An invocation request can be initiated by referring to the destination arobject's id and the operation name from anywhere in the system. Although a single invocation is initiated through the arobject id, multiple invocation can be initiated by referring to a reference name of arobjects which offer the same service. In this case, a calling arobject may continue to perform its activity and receive one or more results by using a *GetReply* primitive.

The communication subsystem also interacts with the transaction subsystem to coordinate the necessary transaction management. For instance, a *Request* primitive is treated as an elementary transaction consisting of three steps: a sending part of the *request*, an invocation linking part, and a receiving part of the *request*. Then, the linking part links the requestee's *Accept*, computation, and *Reply*. Since all message activities in this transaction are defined as compound transactions and the invocation linking part is defined as an elementary transaction, it is possible for the requestee's computation to be a nested elementary or compound transaction of the top-level transaction.

6.4.1 Message Header and Body

A message consists of header and body parts. The header part is not writable from a client and only the body part can be set by a client.

The message header contains the following data:

- Transaction id
- Requestor's arobject id
- Destination arobject id
- Destination operation name
- Number of arguments
- Size of message in bytes

It should be noted that even though message typing is not supported at runtime, type checking of messages between requestor and requestee can be done solely at compile time. Since message communication preserves message boundaries, it does not offer a "stream-oriented" communication interface at this level.

6.4.2 Message Queue

There are two types of message queues associated with arobject and process instances. A *request-queue* is allocated for an instance of an arobject. When a new arobject instance is created, the arobject/process manager creates a request-queue associated with its arobject descriptor. That is, the body of the message queue is kept in the kernel at the running node of the initial process. A *reply-queue* is allocated for an instance of a process. When a process issues an invocation request to an arobject, its results will be placed in the reply-queue.

In both message queues, each entry is represented by a *message descriptor* and can be checked without accepting the message itself.

6.4.3 Communication Manager

When an arobject invocation request is issued at a caller's site, the communication manager sets up a proper header part for the message packet. The message then is sent to the destination arobject. If the destination arobject is in a remote node, the remote invocation protocol is used and the communication manager becomes the monitor for the protocol. Similarly, when a process accesses a

shared private data at a remote node, a remote procedure call is used and the communication manager executes its protocol

6.4.3.1 Components of the Communicatin Manager

Each Communication Manager works with a pair of network I/O workers called, "NetIn" and "NetOut" and a group of "Stub" workers.

NetIn and NetOut workers are resposible to receive and transmit a message packet between two nodes. A Stub worker is used to perform a remote invocation request for a shared privated data object. When a remote invocation is received by the Communication Manager, it assigns the acutual work to the stub worker. Then, the stub worker calls the target procedure with the arguments and returns the result packet to the caller. A sequence of remote invocation is described in Section

6.4.3.2 ArchOS primitives

The following ArchOS primitives are supported for communication management for a client

---

```
trans-id = Request(arobj-id, opr, msg, reply-msg)
trans-id = RequestSingle(arobj-id, opr, msg)
trans-id = RequestAll(arobject-name, opr, msg)
pid = GetReply(trans-id, reply-msg)

(trans-id, requestor, opr) = AcceptAny(acc-opr, msg)
(trans-id, opr) = Accept(requestor, acc-opr, msg)
trans-id = Reply(pid, req-trans-id, reply-msg)

ptr-mds = CheckMessageQ(qtype, selector, selector id)

val = CaptureCommArobject(arobj id, commtype, requestor, req opr)
val = CaptureCommProcess(pid, req opr)
val = WatchCommArobject(arobj-id, commtype, requestor, req opr)
val = WatchCommProcess(pid, req opr)
```

TRANSACTION_ID trans-id
: The transaction id of the transaction on whose behalf the request is being made.

AID arobj id
: The unique id of the receiving arobject.

OPE_SELECTOR opr
: The name of the operation to be performed.

MESSAGE *msg
: A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call by value semantics must be used).

REPLY-MSG *reply-msg
A pointer to the reply message.

AROBJ-REFNAME arobj-refname
The reference name of the receiving arobject(s).

OPE-SELECTOR acc-opr, req-opr
The name of operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the transaction on whose behalf the request is made.

MSG-DESCRIPTORS *prt-mds
Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype, commtype
This indicates either "request-" or "reply-" message queue.

BOOLEAN val TRUE if the primitive was executed properly; otherwise FALSE.

AID requestor The aid of the communicating arobject.

The *Request* primitive provides remote procedure call semantics in which the requesting process invokes an operation by sending a message and blocks until the receiving arobject returns a reply message. Identically, if the receiver arobject is the same arobject, a local operation will be invoked.

The *RequestSingle* and *RequestAll* primitives can send a request message and proceed without waiting for a reply message. The *RequestSingle* primitive provides nonblocking one-to-one communication and, the *RequestAll* primitive supports one-to-many communication. The requesting process may thus invoke an operation on more than one instance of an arobject or process with one request. To receive all of the replies, the *GetReply* primitive may be repeated until a reply with a null body is received.

The *GetReply* primitive receives a reply message which has the specific transaction id generated by the preceding *RequestAll* primitive. If the specific reply message is not available, then the caller will be blocked until the message becomes available.

The process responsible for an arobject operation receives a message using the *Accept* primitive. Using the selection criteria specified, the operating system selects an eligible message from the

arobject's input queue and returns it. The process operates on the message corresponding with a reply when processing has been completed. If no suitable message is in the request message queue, then the caller will block until such a message becomes available.

The *AcceptAny* primitive can receive a message from any arobject instance with any operation (i.e., "ANYOPR") or a specified operation request. The primitive can return the requestor's transaction id, specified operator, and requestor's aid. The *Accept* primitive can receive a message from a specific requestor arobject and returns the requestor's transaction id and the requested operator.

The CheckMessageQ primitive examines the current status of an incoming message queue without blocking the caller process. The primitive must specify a message queue type, either "request-queue" or "reply-queue". The request queue queues all of the non accepted request messages and is allocated for each arobject instance. The reply queue maintains all of the non read reply messages and is assigned to every process instance. A message can be selected based on the sender's arobject id, operation name, and/or transaction id. If more than one argument is given, only messages which satisfy all of the conditions will be returned. If no corresponding message exists in a specified message queue, a "NULL-POINTER" will be returned.

6.4.4 Remote Invocation Protocol

A remote invocation protocol is used to control the invocation of an operation on a remote arobject.

In a simple remote invocation case, two packets will be exchanged between two nodes: one is a *request* packet and the other is *reply* packet. A simplified version of protocol sequence is shown in Figure 6-11.

When a remote request primitive is issued by a client, the base kernel pass its request to the communication manager. Then, the communication manager places a request into an outgoing packet queue and gives it to a "NetOut" worker. The NetOut worker simply initiates the actual output activity over the net. Note that the NetOut worker can be replaced by a simple procedure call within the communicaiton manager if the context switching is significantly large.

When a remote site receives the request packet, the "NetIn" worker of the communication manager places the packet in its incoming packet queue and notify it to the communication manager. If the destination arobject is ready to accept the request, the communication manager moves the actual message to the destination arobject's receiving buffer. Otherwise, the communication manager places the message in the destination arobject's request queue.

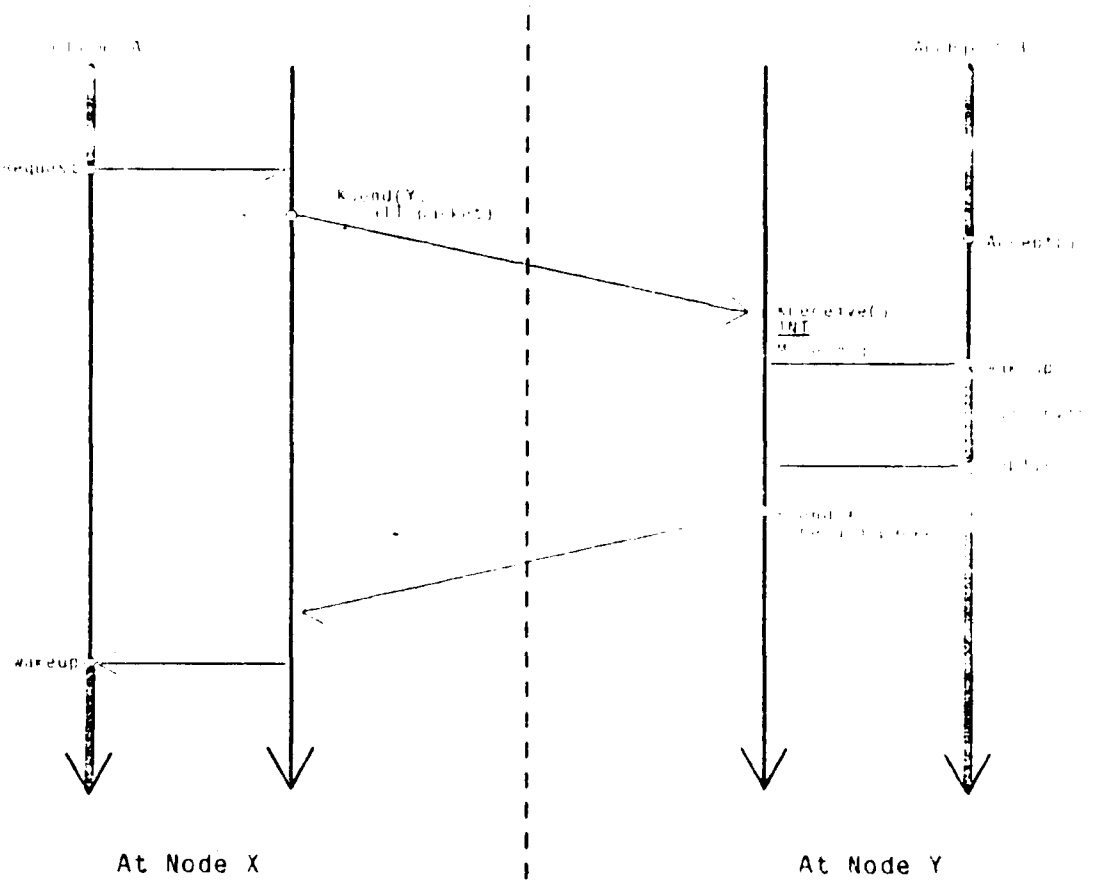


Figure 6-11: Remote Invocation Protocol

6.4.5 RPC for Shared Private Objects

When a client invoke an operation on a shared private object exists on a remote node, the communication manager's *stub* worker performs the actual invocation and returns the result to the caller.

At first, the private object invocation request to a is checked by looking at the shared object table in the shared header segment. If the target object is in a remote node, a remote invocation request, *InvokePrivate* will be sent from the communication manager.

When the remote "NetIn" worker receives the packet, it checks availabiility of the stub workers. If one of stub workers is idle, it assigns the actual invocation request to the stub worker. When the stub worker completes the invocation request, it returns the result message to the communication manager. Then, the communication manager forwards the result packet to the caller.

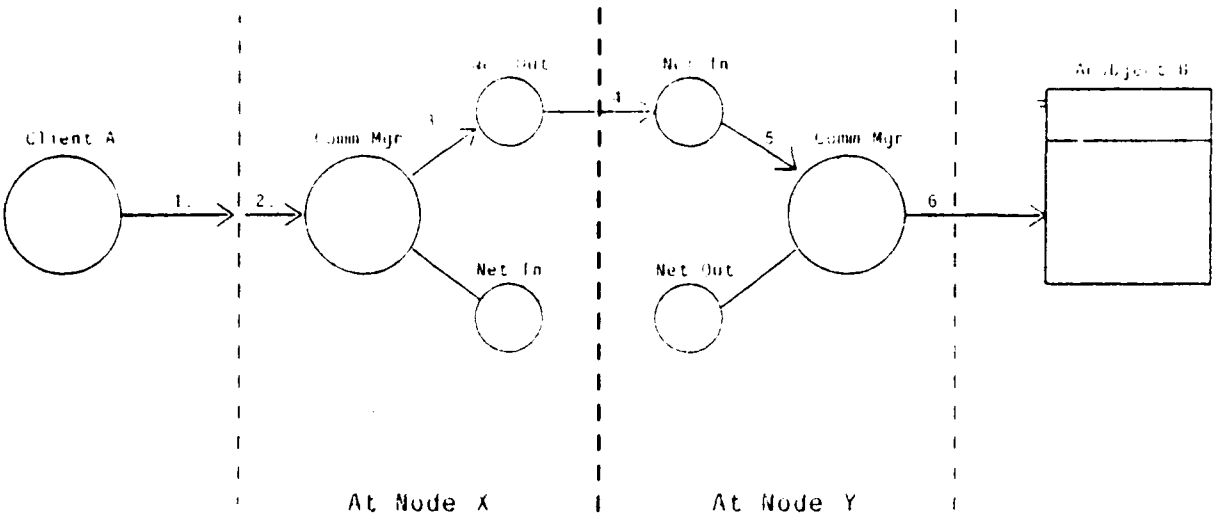


Figure 6-12: Interaction Sequence for a Remote Invocation Request

The basic sequence within in a remote communication manager is shown in Figure 6-13.

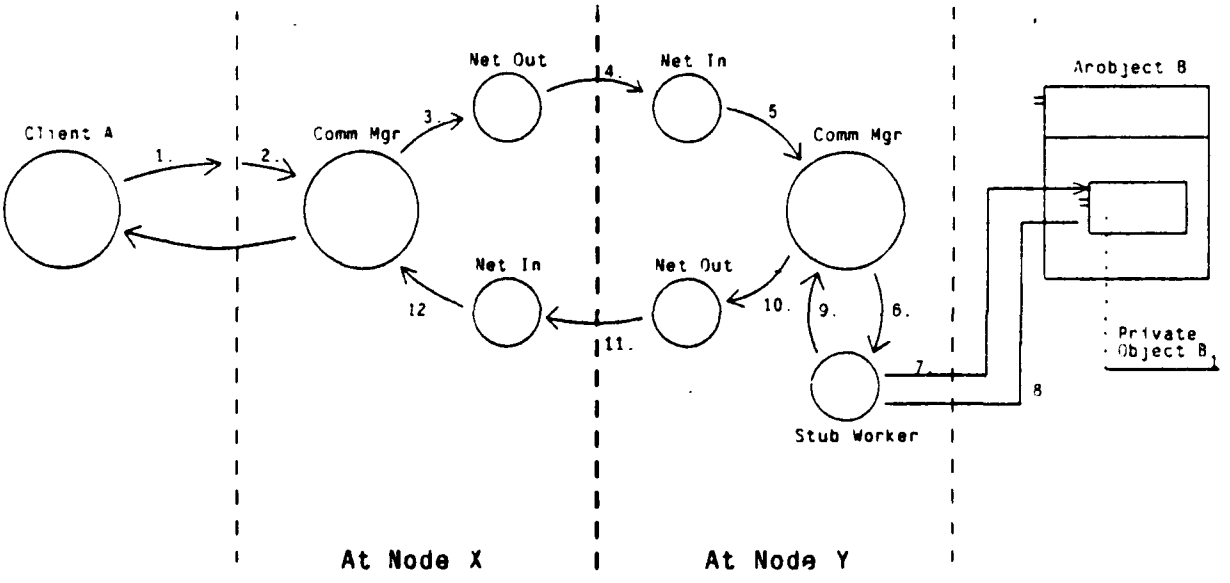


Figure 6-13: Remote Procedure Call Sequence at a Remote Communication Manager

6.5 Transaction Subsystem

The transaction subsystem manages two types of transactions: compound transactions and elementary transactions. The transaction subsystem allows a client to nest a compound transaction or an elementary transaction in any combination. By using the nested elementary transactions, a client can use a traditional "nested transactions" mechanism [Moss 81]. A compound transaction can be used on a "compensatable" atomic arobject. [Sha 85, Tokuda 85] Thus, the transaction subsystem must provide the uniform control mechanisms to perform a "commit" operation for both types and an "undo" operation for the elementary transactions and a "compensation" operation for the compound transactions.

It should be noted that the current transaction subsystem does not provide any mechanisms which support a "cooperating" transaction.

6.5.1 Transaction Types, Scopes, and Tree

- **Transaction Types:**
  A transaction type indicates whether the current transaction is the elementary or compound transaction and the top-level or not.

- **Transaction Scope:**
  A transaction scope is created when a new transaction construct is used in a process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

- **Transaction Tree:**
  A transaction tree is a structure that can be used to trace the dynamic behavior of a set of transactions. The transaction tree also provides information related to the commit protocol and lock propagation among the nested transactions.

- **Transaction Id:**
  The transaction subsystem gets a unique transaction id from the base kernel when a new transaction is initiated and guaranteed to be unique over the lifetime of the transaction. A transaction id is a fixed-length descriptor consisting of a sequence of *parent's node id*, *current node id*, and *local unique id*.

6.5.2 Transaction Management

The transaction manager is responsible for maintaining the consistency of the atomic arobjects even in the case of a node failure. The major activity of the transaction manager is the "BeginTransaction", "Commit" and "Abort" transaction processing.

To commit a transaction, the transaction manager coordinates with the related page set subsystem

and performs atomic update for all atomic objects the transaction has visited. The actual coordination is performed based on the 3 phase commit protocol [Bernstein 83] to improve its reliability.

To abort a transaction, the transaction manager must also coordinate with the page set subsystem, but it performs "undo" for elementary transactions and initiates the necessary "compensation operations" for all pre-committed compound transactions.

6.5.2.1 Components of the Transaction Manager

Each Transaction Manager works with a group of workers, called "Coordinator" and "Subordinator". A "Coordinator" is assigned to keep track the activity of a top-level transaction. When the top-level transaction is committed, the "Coordinator" worker is responsible to perform 3-phase commit protocol among the related subtransactions. A "Suordinator" is used to keep track the activity of a nested transaction.

When a new transaction is created, a transaction descriptor is created in the Transaction Manager and is used to maintain the current status of the transaction and relationship to its child transactions.

6.5.2.2 ArchOS primitives

The following ArchOS primitives are supported for transaction management for a client.

```
tid = BeginTransaction(trantype, timeout)
sval = CommitTransaction(tid)
sval = AbortTransaction(tid)
sval = AbortIncompleteTransaction(req-tid)

tid = SelfTid()
ptid = ParentTid(tid)
trantype = TransactionType(tid)
val = IsCommitted(tid)
val = IsAborted(tid)
```

TIME timeout — The timeout value indicates the maximum lifetime of this elementary transaction.

TRANTYPE trantype — The type of the given transaction, such as "CT", "ET", "Nested CT", or "Nested ET".

TID tid — The id of the specific transaction.

TID ptid — The parent's tid of the given transaction "tid".

INT sval — returns 1 if the requested action was performed on the specified transaction successfully; otherwise returns error staus code.

BOOLEAN val	TRUE if the requested predicate is hold; otherwise returns FALSE.

The *BeginTransaction* primitive creates a transaction descriptor for the requested transaction. The *CommitTransaction* primitive commits the specified transaction.

The *AbortTransaction* primitive aborts the specified transaction and all of its child transactions within the same transaction tree. If the transaction that invokes the *AbortTransaction* primitive does not belong to same the transaction tree as the transaction which is to be aborted, a client cannot abort that transaction. This primitive executes all of the necessary "undo" or "compensate" actions, based on the transaction type, and breaks the current transaction scope. After completion of these actions, the status of all affected atomic objects will be consistent and returned to either "identical" to or "a member of the equivalence class" of their initial (pre-execution) states. The *AbortIncompleteTransaction* primitive also aborts all of the outstanding incomplete transactions which had been initiated by an outstanding *RequestSingle* or *RequestAll* primitive. In other words, all of the nested transactions which belong to the specified request transaction but have not yet completed (committed) will be aborted.

The *SelfTid* primitive returns the id of the current transaction and the *ParentTid* primitive returns the parent transaction id of the given transaction id. The *TransactionType* primitive returns the type of the given transaction (a compound or elementary) and also indicates the transaction level. A *IsCommitted* primitive checks whether the given transaction is already committed or not. A *IsAborted* primitive checks whether the given transaction is already aborted or not.

6.5.3 Three-Phase Commit Protocol

When a top-level transaction is created, its node's transaction manager becomes a primary coordinator to perform the 3-phase commit protocol and the other transaction managers where at least one nested transaction was executed become subordinators. These subordinator transaction managers are also used for backup of the primary manager.

The current 3-phase commit protocol [Bernstein 83] can be summarized as follows:

1. The transaction manager, say TM_i, which is responsible to a top-level transaction, T_{top}, invokes "prepare" operations at all visited node's transaction managers by using a RequestAll primitive. Then, it waits for all transaction managers to acknowledge the "prepare" operation.

2. TM_i invokes "precommit" operations on all the other transaction managers which involve

T_{top}. When the other transaction managers initiate the "precommit" operation, they add a new entry, T_{top}, to its copy of "commit list" of T_{top}. Then, they wait for all acknowledgements for "precommit" to come back.

3. TM$_i$ perform "commit" operations at all related transaction managers.

6.5.4 Compensation Action Management

A transaction manager must initiate a compensation action whenever a compound transaction is aborted and must guarantee that the effects of all committed actions are cancelled out. In general, the ordering relation among the compensation operations is sensitive to satisfy local and global "equivalence relations", thus the transaction manager must maintain a clear ordering rule. In the current algorithm, we take the reverse ordering of the "committed" sequence of the actions.

The book keeping is performed by using a "compensation log" at each arobject. When an arobject invokes a compensatable operation on another arobject as a nested transaction, a "compensation log" record which consists of the current transaction id (i.e., caller's transaction id), arobject name, operation name, parameters, the compensation operation's name, necessary parameters for the compensation operation, will be added on the caller's log. When the caller's transaction is aborted, the system first checks the "cancellation relation" between the current operation and the previous operations by getting the information from the arobject and then the system can simply determine the necessary compensation operations by looking at the compensation log record from the end to the beginning and invoke them.

6.5.5 Lock Management

Proper lock management is necessary to provide a consistent view of atomic data object across the transaction tree.

When a nested elementary transaction commits, all of the locks that it held are passed to the transaction in which it is nested (its parent in the transaction tree); when a nested compound transaction commits, all of its locks are released. The rules that determine which locks a transaction may obtain are more involved. Two transactions are said to be *unrelated* if: (1) they are not contained in a single transaction tree, or (2) they are in a single transaction tree and are concurrently executing siblings or descendants of concurrently executing siblings, or (3) they are in a single transaction tree in which one is an ancestor of the other and either the descendant is a compound transaction or there is a compound transaction on the path connecting the two transaction nodes in the corresponding transaction tree. (Note that according to this definition, a compound child transaction is always unrelated to its parent transaction.)

Two unrelated transactions may compete for locks, and they may hold locks with compatible lock modes for a single data object at any given time. However, if they request incompatible lock modes for a single data object, then one of the competitors will obtain a lock and the other will block until it can receive the desired lock, or it will return to the requestor with an appropriate status indication.

Two transactions are said to be *related* if they are contained in a single transaction tree where one is the descendant of the other, the descendant is an elementary transaction, and there are no compound transactions in the path connecting their respective nodes in the transaction tree. The lock compatibility rules for related transactions are different than those for unrelated transactions. In this case, the descendant transaction can obtain any lock mode for any lock held by a related ancestor in the transaction tree, including incompatible lock modes that would not be allowed if the transactions were unrelated. (Of course, the descendant transaction will have to compete with all of the unrelated transactions in the system to successfully obtain the requested lock with the desired mode.)

The following ArchOS primitives are supported for lock management for a client:

---

newlock-id = **CreateLock**([parent-lockid])
val = **DeleteLock**(lockid)

sval = **SetLock**(lock-type, lockid, lock-mode)
tval = **TestandSetLock**(lock-type, lockid, lock-mode)
tval = **TestLock**(lock-type, lockid, lock-mode)
rval = **ReleaseLock**(lock-type, lockid, lock-mode)

INT sval — 1 if the specified lock is set; 0 if the lock is not set. A negative value will be returned if an error occured.

INT tval — 1 if the specified lock is being held; 0 if the lock is not being held. A negative value will be returned if an error occured.

INT rval — 1 if the specified lock is released; 0 if the lock is not released. A negative value will be returned if an error occured.

LOCK-TYPE lock-type
The lock type can be either "TREE" or "DISCRETE".

LOCK-ID lockid — The lockid indicates the unique id of a lock.

LOCK-MODE lock-mode
The lock mode can be "READ", "WRITE", etc.

The *SetLock* primitive sets a "tree type" or "discrete-type" lock on arbitrary objects by specifying a lock key and its mode. If a requested lock is being held, the caller will block until it is released. The *TestandSetLock* primitive also tries to set a lock, however, it will return a "FALSE" if the lock is being held. If the requested lock is a tree-lock type, then the *SetLock* and *TestandSetLock* primitives may also fail due to the violation of the tree-lock convention (See Section 4.2.4.2).

The *TestLock* primitive checks the availability of a specified lock with a lock mode. In the case of a tree lock, it also checks whether the locking would be legal in the corresponding lock tree. The *ReleaseLock* primitive can release the lock on an object which was gained by the *SetLock* or *TestandSetLock* primitive explicitly.

6.5.6 Recovery Management

Since automatic recovery of application programs is not an easy task for the transaction subsystem, the subsystem guarantees only the consistency of atomic arobjects and provides a handle to distinguish the pre-crash and post-crash situation. Each arobject designer must define a proper action for recovery and let the "INITIAL" process handle a detailed recover sequence.

In ArchOS, each arobject has an atomic variable, called the "restart-counter" which becomes *incremented whenever its host machine restarts. By using the restart counter, the* INITIAL process can examine whether the program is running before a crash or not.

To clean up and retrieve the necessary atomic arobjects, the transaction subsystem must coordinate with the page set manager. In particular, the atomic page set manager can recover the nessary page set for a given atomic arobject.

6.6 File Management Subsystem

The ArchOS File Management Subsystem provides a system-wide, location independent file access service. It provides three types of files: *normal*, *permanent*, and *atomic*. *Normal* files are temporary, and not recoverable following a crash. *Permanent* files are saved on disk, and most closely resemble files on other systems. *Atomic* files are guaranteed to be failure atomic under "soft and clean" failures [Bernstein 83]. All three types of files are client level arobjects when active (i.e. open). When inactive, permanent and atomic files are saved using page sets (See Section 6.7)

The implementation of files as regular client level arobjects achieves two main advantages. First, the full power of the ArchOS transaction facilities is available, permitting the arbitrary nesting of

atomic file operations within other nested transactions. Second, the file arobjects are treated in the same manner as other arobjects by the Time driven Scheduler and the Time driven Virtual Memory Manager. This ensures that critical files (data) will be available (scheduled and paged in) when they are to be accessed by critical processes, thus reducing the number of missed [illegible] times.

The File Subsystem uses logical, location independent names for files. Hence, files are not constrained to reside on particular disk volumes. They can be dynamically moved to other disks by the system, if desired, for improved global disk usage. The file name space is flat (not hierarchical), but facilities are provided to allow many of the benefits of hierarchical directories. File names are expected to be relatively long strings composed of several shorter strings (called *components*), separated by the special character "/" (slash). An initial substring of a name (up to a slash) is known as a *prefix*.

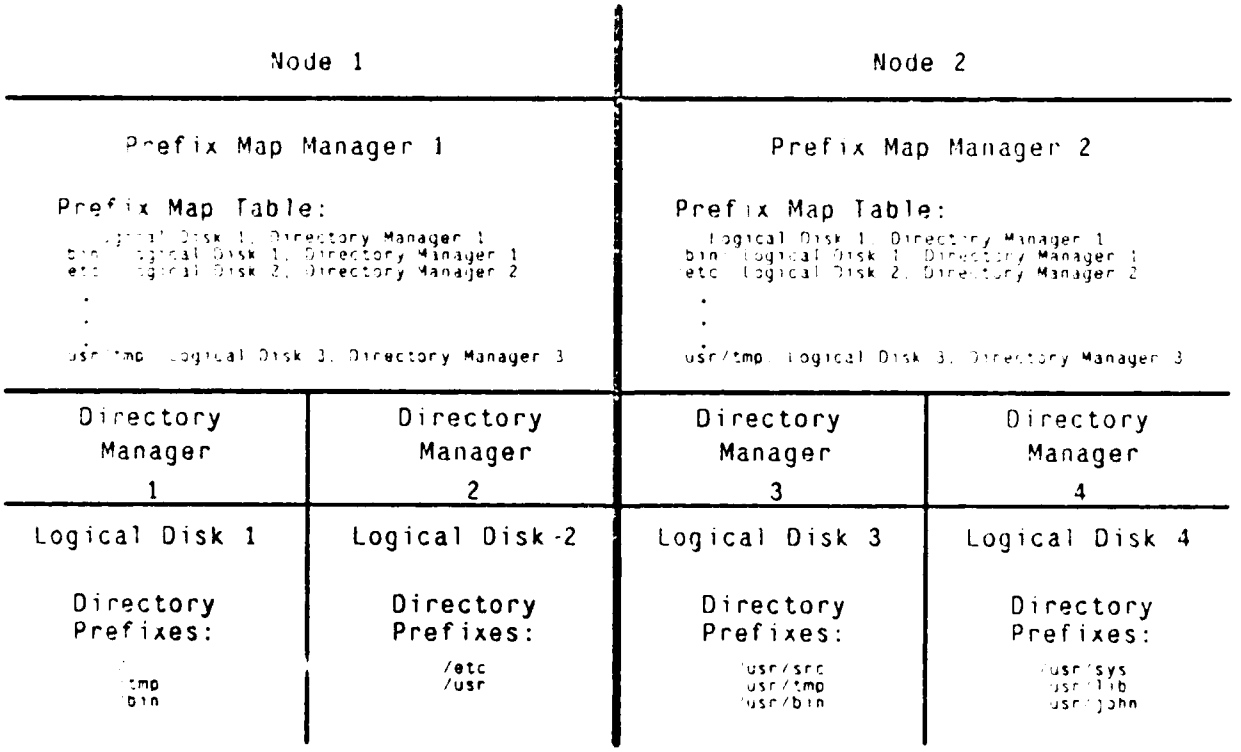


Figure 6-14: The Partitioned File Subsystem Directory Structure

The system-wide file system directory is partitioned, where each partition is saved in a known location (refer Figure 6-14). For example, the part of the directory containing all file names with a particular prefix is saved at one logical disk. This style of implementation was dictated by efficiency

considerations, such as in creating and locating files. Facilities are provided for moving parts of the directory from one disk to another, but this is expected to occur infrequently

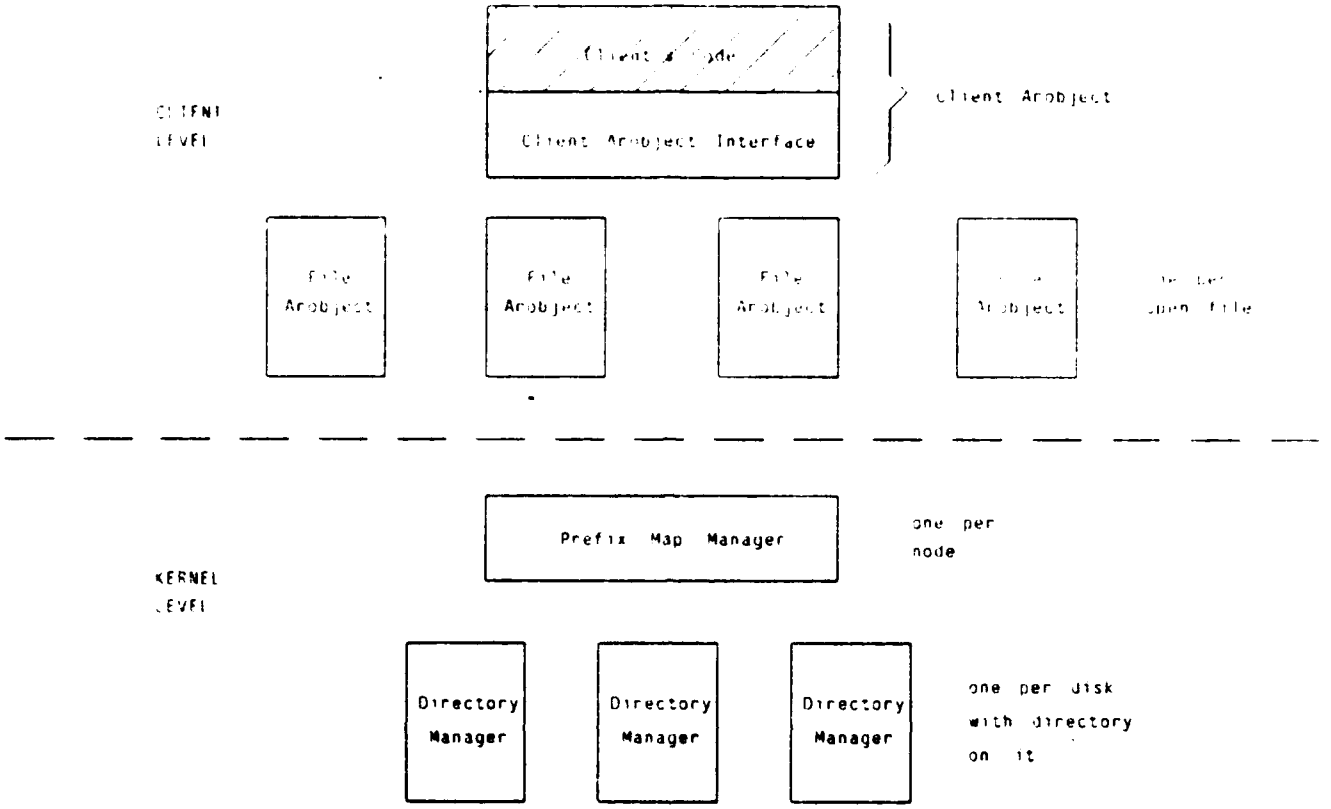


Figure 6-15: Subsystems within the File Management Subsystem

The functionality of the File Management Subsystem is provided by four separate but co-operating entities (refer Figure 6-15).

1. Each client arobject is linked with a standard file system interface. This interface consists of a library of routines, which hide the implementation details of the other subsystems within the File Management Subsystem.

2. Each open (active) file is represented by an instance of a user level file type arobject.

3. Each node in the system contains a kernel level Prefix Map Management Arobject, which is responsible for mapping file name prefixes to the logical disks on which their directory entries reside.

4. Each logical disk in the system containing a portion of the directory has an associated kernel level Directory Management Arobject.

The above four entities will be described in some detail in the following sections.

6.6.1 Client Arobject's File System Interface

A client arobject will always interact with the file system through the use of a standard set of library routines. These routines are together referred to as the Client Arobject's File System Interface (or the CA Interface). The primary purpose of the CA Interface is to transparently interact with the three other subsystems of the file system (File Arobjects, Prefix Map Manager, and Directory Manager) in supporting client level functionality. Thus, it hides the details of the interactions with the other parts of the File Subsystem, and provides a cleaner interface than the raw system primitives.

A second aspect of the CA Interface's functionality is that it provides appropriate internal buffering to improve efficiency. For open files, it also keeps track of the sizes of the files.

The following list of primitives is intended to illustrate the functionality of the ArchOS standard file I/O library. It is not an exhaustive list. Many other primitives can be added to make file management more convenient for the client.

---

fid = **CreateFile**(filename, filetype, mode [, node])
val = **DeleteFile**(filename)
fid = **OpenFile**(filename, mode)
val = **CloseFile**(fid)
val = **RenameFile**(oldname, newname)
val = **SyncDirectory**(filename)

val = **SetPrefix**(prefix)
prefix = **GetPrefix**()

nread = **ReadFile**(fid, buffer, length)
nwritten = **WriteFile**(fid, buffer, length)
nwritten = **ZeroFile**(fid, length)
val = **SeekFile**(fid, byteoffset, origin)
val = **SyncFile**(fid)

val = **StatusFile**(fid, statusbuffer)
val = **InfoFile**(filename, infobuffer)

INT fid — The ID for the file.

BOOLEAN val — TRUE if the specified operation is done successfully; otherwise FALSE.

FILENAME prefix — The filename prefix currently in use, which will be added to any filename tails being specified.

INT nread — The number of bytes actually read.

INT nwritten The number of bytes actually written.

FILENAME filename
The name of the file for which the specified operation is to be performed.

FILETYPE filetype The type of the file to be created: NORMAL, PERMANENT, or ATOMIC.

MODE mode One of four possible modes in which the file can be opened: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE.

NODENAME node The ID of the preferred node on which the file should be created.

FILENAME oldname, newname
The old and new names (respectively) of the file being renamed.

BUFFER *buffer The address for the data buffer.

INT length The number of bytes to be read or written.

INT byteoffset The position in number of bytes from the origin.

FILEORIGIN origin Starting position in the file, which can have one of three values: START-OF-FILE, END-OF-FILE, and CURRENT-POSITION.

FSTATUSBUFFER *statusbuffer
The buffer address for returning dynamic file status information.

FINFOBUFFER *infobuffer
The buffer address for returning static file information.

The CreateFile primitive creates a file of the specified type (NORMAL, PERMANENT, or ATOMIC) on the preferred computing node. If a node preference is not specified, the file is created on any node. The newly created file is then opened in the specified mode. The *DeleteFile* primitive deletes the specified file. *OpenFile* opens the specified file in one of four modes: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE. The *CloseFile* primitive closes the specified file.[29] *RenameFile* changes the name of the file from the old name to the new name specified. The *SyncDirectory* primitive is used for saving any buffered directory information for the specified file on disk.

The *SetPrefix* primitive provides a short hand technique for giving filenames. A file prefix can be

---

[29] All open files for a client are registered with the kernel by the CA Interface. Hence, if an arobject with open files is destroyed the kernel can close all these open files. The occurrence of each of the three primitives *CreateFile*, *OpenFile*, and *CloseFile*, is reported to the kernel.

specified, which is automatically concatenated with filename tails to form the complete file name. The *GetPrefix* primitive allows the client to obtain the current value of the file prefix.

The *ReadFile* primitive is used for reading the specified number of bytes from a file, starting at the current position. The bytes read are returned in a buffer in the client's address space. Similarly, the *WriteFile* primitive writes the specified number of bytes from the buffer, at the current position in the file. The *ZeroFile* primitive is used for writing the specified number of zero bytes, starting at the current position in the file. The ability to zero a file is useful when truncating files, and creating sparse files. The *SeekFile* primitive allows random access to a particular position in the file. The position can be specified as a byte offset from the start or end of the file, or from the current position in the file. The *SyncFile* primitive flushes the contents of a file from buffers in memory onto disk.

The *StatusFile* primitive is used for obtaining dynamic status information of a file in the buffer provided. Information such as the current mode of file access, and the number of active and outstanding open requests on the file is provided. The *InfoFile* primitive, on the other hand, provides static file information, such as creation date, last modification date, length of file, etc.

6.6.2 File Arobjects

Each open file in the system is represented by a client level file arobject. There are three different types of file arobjects, corresponding to the three different types of files: normal, permanent, and atomic. A file arobject maintains the "file buffer", and provides byte level file I/O. The file is mapped onto the virtual address space of the file arobject, and the buffer is maintained automatically by the Virtual Memory Subsystem. The two main functions of a file arobject are: (1) maintaining locks, and thus ensuring consistent concurrent access to the file, and (2) handling *read* and *write* operations on the file. Locks are set on the entire file. When the file is opened, the type of access is specified, and the appropriate lock is set. The lock is released only when the file is closed. In addition to lock management, reading and writing, a few other primitives such as *FASync* and *FAStatus* are also provided.

```
filesize = FAOpen(mode)
val = FAClose()

nbr = FARead(location, nbytes, buffer)
nbw = FAWrite(location, nbytes, buffer)
nbw = FAZero(location, nbytes)

val = FASync()
val = FAStatus(statusbuffer)

val = FARestart(fdm-aid, filesize)
```

INT filesize	The size of the file in bytes.
BOOLEAN val	TRUE if the specified operation is done successfully; otherwise FALSE.
INT nbr	The actual number of bytes read.
INT nbw	The actual number of bytes written.
MODE mode	One of four possible modes in which the file can be opened: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE.
INT location	The location (in bytes from the beginning of the file) within the file at which reading or writing starts.
INT nbytes	The number of bytes to be read or written.
BUFFER *buffer	The address for the data buffer.
FASTATUSBUFFER *statusbuffer	The buffer address for returning status information.
AID fdm-aid	The ID of the Directory Management arobject, to be used when closing the file.

The *FAOpen* primitive allows a file to be opened in one of four possible modes: "READ", "WRITE", "READ-ONLY", and "EXCLUSIVE-WRITE". These modes apply to all three types of files, but the behavior of a mode does depend on the type of file. Once the file is opened in a particular mode, the appropriate type of lock is set on it to ensure consistency. The size of the file is returned to the client arobject. The file size, as well as the current position pointer are maintained by the Client Arobject Interface. The *FAClose* primitive closes the file. It also deactivates the file arobject if there are no other outstanding "opens" on the file.

The *FARead* and *FAWrite* operations allow multiple bytes to be read or written at a time. The current position at which reading or writing should start, and the number of bytes to be read or written are provided, along with a pointer to the data buffer[30]. The actual number of bytes read or written is returned by the operation. The *FAZero* primitive allows a number of bytes inside of the file to be zeroed out. Thus, it is possible to truncate a file, or maintain a sparse file in this system.

The *FASync* operation flushes the contents of the file buffer onto the disk, so that any buffered information is synchronized with the copy of the file stored on disk. The *FAStatus* primitive returns dynamic file information in the statusbuffer provided. Information is provided about the mode of file access, the lock compatibility of open files, and the number of active and outstanding open requests.

The *FARestart* operation is used for initialization purposes when the file arobject is first created (which can happen when the file is created, or first opened). It is invoked by an instance of the Directory Management arobject, which provides its own ID and the current size of the file as parameters. The ID of the Directory Management arobject is used by the file arobject when the file is closed by any client.

6.6.2.1 Components of a File Arobject

The components of a File Arobject are described below and shown in Figure 6-16. Each file arobject has a single process called the *FA Manager Process*, which receives all requests for operations, carries out the appropriate operations, and returns the results. It operates on three main data structures: (1) the Open Client List (OCL), (2) the Request Client List (RCL), and (3) the file buffer. The *Open Client List* maintains a list of the clients which have successfully opened this file, and the mode of access for each client. Since locking is done on a per file basis, all these clients must have compatible locking on the file. The *Request Client List* is a list of the clients waiting for their open operations to be completed. The information maintained for each request is a triple consisting of the request transaction ID, the ID of the requesting client arobject, and the mode of access requested.

The file buffer is a part of the file arobject's address space. It is maintained in a number of fairly large chunks (32 KBytes per chunk[31]). A list of pointers to these chunks and a count of the number of chunks are also maintained. The advantage of large chunks is that a long list of chunk pointers is not needed, and hence searching for a particular page in the file can be more efficient.

---

[30] If the client is remote with respect to the file arobject, the communication mechanism automatically handles data transfer to the remote buffer.

[31] The size of the chunks is determined by the architecture of the SUN workstations, the current target machines for ArchOS implementation.

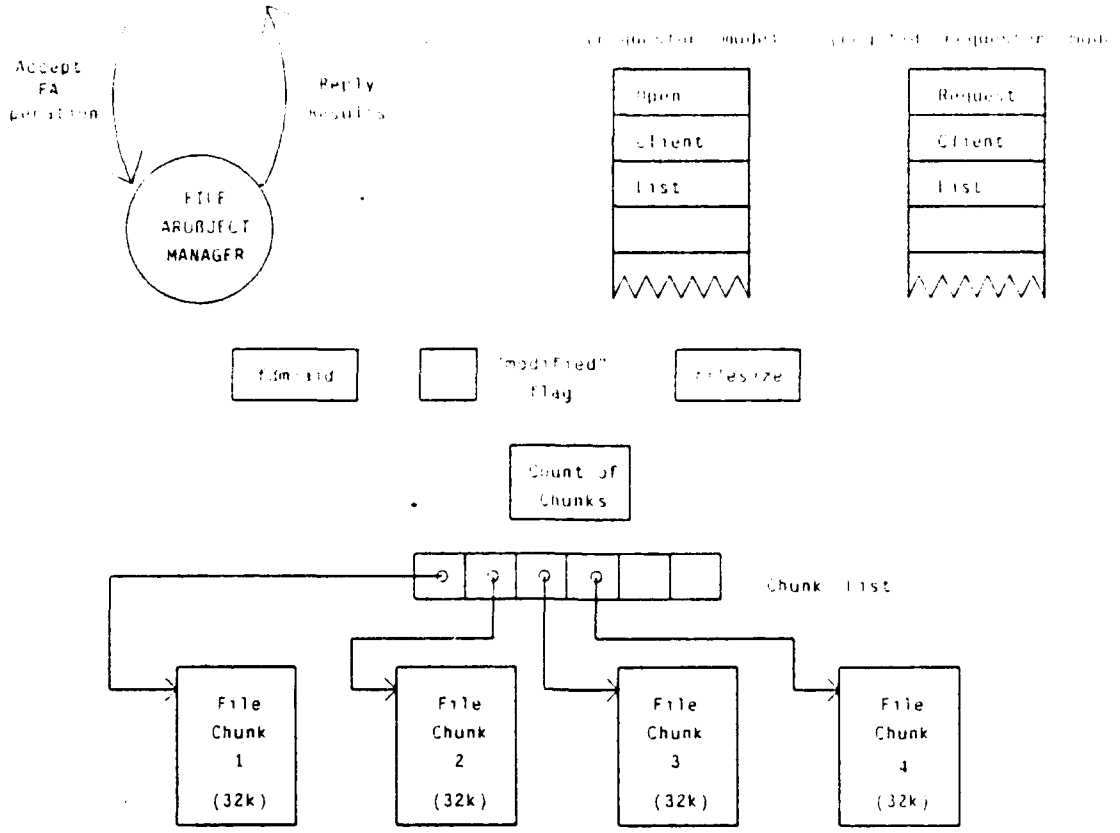


Figure 6-16: Components of a File Arobject

In addition to the three main data structures, a few other pieces of information are also maintained. Some of these are the ID for the Directory Management arobject, a flag which indicates whether the file has been modified, and a variable which saves the current size of the file[32].

6.6.2.2 Normal and Permanent File Manipulations

Opening of files is allowed to proceed on a FCFS basis[33], but multiple compatible opens are permitted. When a *FAOpen* request is received by the FA Manager process, the Open Client List is first checked to see whether there are other ongoing requests. If the OCL is empty, the incoming request is placed on that list. If the Request Client List has some waiting clients, then the new request

[32] The size of the file saved here is updated whenever *FAWrite* and *FAZero* operations are invoked. This is not the most up-to-date value, since the Client Arobject Interface buffers several read and write operations.

[33] The policy for opening files can possibly be set by the user to better suit the constraints of the real time application.

is added to that list. If the RCL is empty, and the *mode* of the incoming request is compatible with the modes of the ongoing requests in the OCL, the new request is added to the OCL, but if the mode is incompatible, then it is added to the RCL. Once the incoming request has been added to the OCL, the *FAOpen* is complete, and the client is signalled. If the incoming request is added to the RCL, the *FAOpen* is suspended, pending lock availability.

When the *FAClose* operation is requested, the OCL is first checked for the requesting client, and the corresponding entry is removed. Next, the FDClose() operation is invoked for the appropriate Directory Management (FDM) arobject. The FDM arobject maintains a count of outstanding open requests, and informs the File Arobject whether it should continue, deactivate, or kill itself. If the FA has to continue, it moves all of the compatible entries from the head of the RCL to the OCL, and notifies the corresponding clients. If there are no outstanding open requests, the FA deactivates itself. If the file has been deleted, and the last *FAClose* operation has completed, the FA kills itself.

The reading and writing of normal and permanent files is very straightforward. The system "copy" mechanism carries out the transfer of data between the FA address space and the client's address space. Reading and writing to disk is automatically handled by the virtual memory manager. Each time the *FAWrite* and *FAZero* operations are invoked, the file arobject updates its notion of the current file size.

6.6.2.3 Atomic File Manipulations

The operations on atomic files are quite similar to their counterparts for normal and permanent files. The main difference is that atomicity of the operations has to be guaranteed whenever the file is manipulated. The *FAOpen* and *FAClose* operations are the same as described in the previous section, since these operations only affect the OCL and RCL data structures, and do not touch the atomic file data.

The *FARead* and *FAWrite* operations are implemented as elementary transactions, so that they can be arbitrarily nested inside of other transactions. The *FARead* operation invokes the *Copy* mechanism for copying the data from the file arobject's address space to the client's address space. For the *FAWrite* operation however, the *AtomicCopy* operation is invoked, which updates the file in the main memory, and also propagates the write operation to the Atomic Page Set Subsystem (refer Section 6.7.2).

6.6.3 Prefix Map Management

The directory of the ArchOS File Subsystem is distributed across multiple nodes and multiple disks of the system. Specifically, all directory entries with a particular prefix are on a particular logical disk. Hence, we need to maintain a system wide table which can map the directory fragments corresponding to different prefixes, to the logical disks on which these fragments reside. The main function of the File Prefix Map Management Subsystem (or FPMM subsystem) is to maintain this mapping in a table known as the Prefix Map Table (or PMT). There is one instance of the FPMM subsystem at the kernel level of each node of the distributed system. A copy of the entire system wide Prefix Map Table is maintained by each FPMM instance.

All of the functionality provided by the FPMM subsystem is related to mapping file name prefixes to IDs of appropriate Directory Management Arobjects, which save the directory entries for those prefixes. Once the appropriate Directory Mangement Arobject for a file has been determined, all future operations on that file are performed either by the File Arobject, or by the Directory Mangement Arobject (depending on the operation in question). The FPMM subsystem, on the other hand, provides operations for accessing and maintaining the Prefix Map Table, e.g. at restart, and when disk volumes are mounted and unmounted. The ability to create and delete new directory fragments by adding or removing file prefixes is also provided.

---

fdm-aid = **FPMap**(filename)

val = **FPRestart**()
val = **FPMount**(diskid)
val = **FPUnmount**(diskid)
val = **FPAssign**(prefix, diskid)
val = **FPUnassign**(prefix)

val = **FPInsertTable**(prefix, diskid, fdm-aid)
val = **FPRemoveTable**(prefix)
val = **FPRequestTable**(tablebuffer)

AID fdm-aid — The arobject ID of the Directory Management subsystem.

BOOLEAN val — TRUE if the specified operation is done successfully, otherwise FALSE.

FILENAME filename — The name of the file for which the specified operation is to be performed.

DISKID diskid — The Logical Disk ID of the disk volume being operated on.

FILENAME prefix The filename prefix being used in the specified operation.

TABLEBUFFER *tablebuffer
The buffer used for returning the contents of the Prefix Map Table.

The *FPMap* primitive returns the arobject ID of the Directory Mangement Subsystem instance which manages the directory fragment for the given file. This primitive will usually be invoked by the Client Arobject Interface, when the first operation (e.g. *CreateFile* or *OpenFile*) is requested by a client. Once the appropriate Directory Management Arobject for a specific file has been determined, any further requests relatd to that file will not be addressed to the FPMM Subsystem.

The main purpose of the *FPRestart* primitive is to reconstruct the Prefix Map Table. This is done by determining which disks are mounted on this node, and by acquiring information from the PMTs residing at other nodes in the system. The *FPMount* primitive is used when a disk volume is mounted. If the new disk has a portion of the directory on it, the Prefix Map Table is modified to reflect this, and an FDM arobject is created to manage that directory. The *FPUnmount* primitive is used when a disk volume is removed. Any entries corresponding to this disk in the Prefix Map Table are removed, and the FDM arobject is informed.

The *FPAssign* and *FPUnassign* primitives are used for adding and removing prefixes to and from the entire file system directory. A new prefix can be added to the specified disk irrespective of whether the disk already has a part of the directory on it or not. When a new prefix is added, modifications are made to the Prefix Map Table and to the disk itself. An FDM arobject for the disk also has to be created, if it does not already exist. The *FPUnassign* primitive removes a prefix from a disk by making modifications to the PMT and to the disk. It also informs the FDM arobject if necessary. This primitive will execute only if the prefix being unassigned does not correspond to any existing files; otherwise an error indication is returned.

Three primitives are provided for obtaining and modifying information from the Prefix Map Table. These primitives will be invoked primarily by FPMM arobjects on other nodes of the system. The *FPInsertTable* primitive inserts a new entry into the PMT, which consists of the directory prefix being added, the ID of the logical disk on which the directory portion exists, and the ID of the FDM arobject responsible for the management of that directory portion. The *FPRemoveTable* primitive removes the specified prefix entry from the PMT. The *FPRequestTable* primitive asks for all the contents of the PMT to be returned in the buffer provided.

6.6.3.1 Components of the Prefix Map Management Subsystem

The FPMM subsystem consists of three components: (1) the Prefix Map Table or the PMT, (2) the Prefix Map Manager process, and (3) the Prefix Map Worker process (refer Figure 6-17). The PMT keeps a copy of the entire system wide mapping between prefixes and directory locations. Its entries are a series of triples consisting of the file name prefix, the ID of the logical disk containing the directory fragment corresponding to this prefix, and the ID of the FDM arobject which manages this directory fragment. It is important to ensure the consistency of the multiple copies of the PMT.

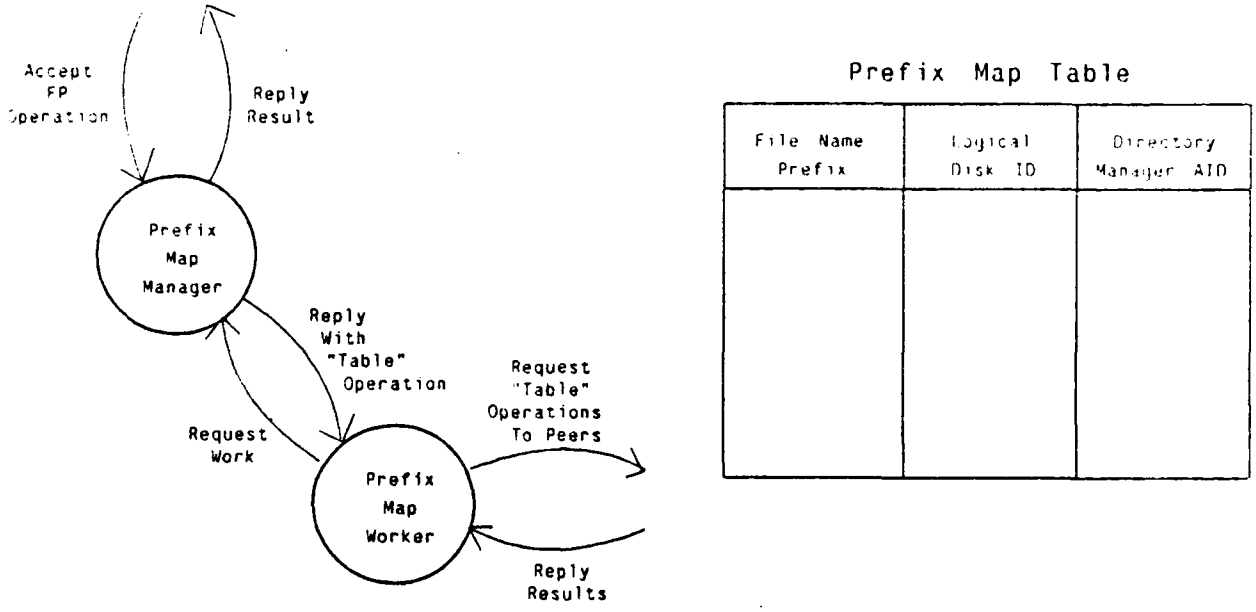


Figure 6-17: Components of the Prefix Map Management Subsystem

The Prefix Map Manager process is responsible for providing most of the functionality of the FPMM subsystem. It accepts all requests for FPMM operations, and returns the results. The only purpose of the Prefix Map Worker process is to wait on behalf of the Manager process when peer level primitives are invoked by the Manager process. These primitives (FPInsertTable, FPRemoveTable, FPRequestTable) are used for maintaining system wide consistency between the PMT instances. In the absence of the Worker process, deadlocks can arise if multiple peer operations are in progress concurrently.

6.6.3.2 Directory Mounting and Reassignment

In order to understand the implementation of the primitives *FPMount*, *FPUnmount*, *FPAssign*, and *FPUnassign*, it is important to have some information about the layout of the logical disk volumes. There is a special page (Page 0) on each disk volume which contains information about the contents and location of several important data structures stored on the disk. Hence, by examining Page 0, it is possible to determine whether there are any directory fragments on the disk, and if so, what file prefixes they correspond to.

When a logical disk is mounted on a node, the *FPMount* primitive is invoked on that node to update the PMT if necessary. First, Page 0 of the disk is checked to see whether the disk has a directory fragment on it, and if so, to determine the prefixes which are mapped on it. All these prefixes are then entered in the local copy of the PMT, along with the ID of the logical disk. An FDM arobject is created and initialized to manage the newly mounted disk directory, and its arobject ID is entered in the PMT. This completes the updating of the local PMT. All newly added entries are now sent to all other instances of FPMM by invoking the *FPInsertTable* primitive on peer FPMM arobjects.

When a disk is unmounted, the *FPUnmount* primitive is invoked. If there are any entries in the PMT which correspond to this logical disk, these entries are removed. The FDM arobject for that disk is cleaned up and destroyed by invoking the *FDUnmount* primitive. Furthermore, peer FPMM arobjects are informed by invoking the *FPRemoveTable* primitive.

The *FPAssign* primitive assigns a new prefix to a particular logical disk. First Page 0 of the disk is checked to see whether it already has a directory on it or not. If there is no directory, a directory root page set is created on that disk, and a pointer to its root page is entered in Page 0, along with the new prefix being added. An FDM arobject also has to be created to manage the directory on this disk. An entry is added to the PMT corresponding to the new prefix, logical disk, and FDM arobject The addition of a new prefix to a disk which already has a directory fragment on it is much simpler. It only requires a new entry on Page 0 of the disk and in the PMT, since the directory root page set and FDM arobject are already present. In both cases, once the PMT has been updated, the peer FPMMs are informed of the new entry.

The *FPUnassign* primitive removes a directory segment (corresponding to a prefix) from a disk. However, this operation is only allowed when there are no files in the system which correspond to that prefix. First, the prefix is removed from Page 0 of the disk. If, as a result of this removal, there are no more prefixes left on the disk, the directory root page set also has to be freed. If the disk has no directory segment left on it, the FDM arobject for the disk is destroyed by invoking *FDUnmount*. In

any event, the entry for the prefix is removed from the PMT, and the peer FPMMs are informed of the removal.

6.6.3.3 Restart

The main purpose of the *FPRestart* primitive is to recreate the Prefix Map Table. To do this, the Manager process first invokes the Worker process to obtain the PMT (with FPRequestTable) from one of the other nodes in the system. In the meantime, the Manager itself checks the Mount Table of its node (see Section 6.7.4) to determine all the logical disks on the node. For each mounted disk, it essentially executes the functionality of the *FPMount* primitive. It checks Page 0 for prefixes, enters these in the PMT, and creates an FDM arobject for each disk with a directory fragment. Once all entries are made to the PMT, all peer FPMMs are informed of the new entries.

There was a danger of deadlocks in the restart sequence, especially if multiple nodes happened to come up at the same time. To avoid deadlock, a separate Worker process is provided, which waits on the peers FPMMs. In addition, a timeout is associated with the *FPRequestTable* primitive to avoid indefinite blocking. If several nodes come up at the same time, each node can incorporate entries in its PMT pertaining to its own logical disks, and then send this information to the other nodes.

6.6.4 Directory Management

The main purpose of the Directory Management Subsystem is to maintain a part of the file directory, and thereby map filenames occurring in this directory fragment to their corresponding global page set identifiers. This mapping determines the logical disk on which the file resides, and the ID of the page set arobject which manages the root page set for the file. In addition to the mapping information, the directory also maintains static file information, such as creation date, modification date, and file length. An instance of the File Directory Management Subsystem (FDM Subsystem) exists for each disk with a directory fragment on it.

Operations which require directory mapping information, such as *FDOpen*, *FDClose*, *FDCreate*, and *FDDelete*, are all supported by the FDM subsystem. Filename matching operations, such as finding all files with a matching prefix, are also directory oriented operations, and are performed by this subsystem. In addition to these operations, primitives are provided for accessing and modifying the directory entries.

```
fa-aid = FDOpen(filename)
fa-aid = FDCreate(filename, type [, node])
action = FDClose(modified, filesize)
val = FDDelete(filename)
val = FDUndeleteAtomic(filename)
val = FDExpungeAtomic(filename)

val = FDSync([filename])
last = FDFind(prefix, after, comp, buffer, bufsize)

val = FDInsert(filename, file-info)
val = FDGet(filename, file-info)
val = FDRemove(filename)

val = FDRestart(diskid)
val = FDUnmount()
```

AID fa-aid — The ID of the File Arobject, corresponding to the file being opened or created.

FDACTION action — Specifies one of three possible actions to be taken: CONTINUE, DEACTIVATE, or KILL.

BOOLEAN val — *TRUE if the specified operation* is done successfully, otherwise FALSE.

FILENAME last — A number of components or tails (depending on the value of *comp*) of filenames are found, matching the given prefix. The last component or tail is returned in the *last* variable. If the end of the list of components or tails has been reached, a special value (NULL) is returned.

FILENAME filename — The name of the file for which the specified operation is to be performed.

FILETYPE type — The type of file to be created: NORMAL, PERMANENT, or ATOMIC.

NODENAME node — The ID of the preferred node on which the file should be created.

BOOLEAN modified — Flag which specifies whether the file has been modified or not.

INT filesize — The size of the file in bytes.

FILENAME prefix — The filename prefix which has to be matched, and corresponding to which all filename components or tails of filenames have to be returned.

FILENAME after — The value returned by the variable *last* is used here. Hence it is either a component or tail of a filename. Matching of components or tails has to start after this element. If the value of *after* is NULL, matching starts from the first element.

BOOLEAN comp	If the value is TRUE, only the next components of the filenames following the matching prefix are returned. If the value is FALSE, the entire tails of filenames are returned.
BUFFER *buffer	The address for the data buffer.
INT bufsize	The size of the buffer being provided.
FILEINFO file-info	Returns all the information about a file saved in the directory entry.
DISKID diskid	The Logical Disk ID of the disk volume being operated on.

The *FDOpen* primitive opens an existing file. If this is the first *FDOpen* operation on the file, the Arobject/Process Management Subsystem is called upon to activate the inactive file arobject, and return the ID of the active file arobject. If the file has already been opened, the count of opens is incremented, and the ID of the active file arobject is returned. If the file does not exist, an error is returned. The *FDCreate* primitive creates a file of the specified type (NORMAL, PERMANENT, or ATOMIC) on the preferred computing node, if possible. The Arobject/Process Management Subsystem is requested to create a file arobject of the appropriate type, and the ID of this arobject is returned. A directory entry for the newly created file is added. Following creation, the file is opened. If the file already exists, an error is returned.

The *FDClose* primitive is always invoked by the file arobject, when some client has requested the *FAClose* operation. It responds by specifying one of three operations (CONTINUE, DEACTIVATE, or KILL), to be carried out by the file arobject. It also modifies the static file information saved in the directory. If the file has been modified, the modification date and time is changed to the present time, and the file length is updated.

The actions taken by the *FDDelete* primitive depend on whether the file is open or not at the time of the request. If the file arobject is inactive, *FDDelete* deletes the file and removes its directory entry. If the file is open at the time the *FDDelete* request is received, a clean file deletion is provided. Further *FDOpen* operations are not allowed, and once the file has been closed by all current users, it is deleted. In the case of atomic files, the *FDDelete* primitive merely flags the file (in the directory) as deleted, so that it can be recovered in case the transaction is aborted.

The *FDUndeleteAtomic* and *FDExpungeAtomic* primitives are provided to allow atomic deletion of files. The *FDDelete* primitive for an atomic file sets a flag in the directory entry. After the transaction

commits, the garbage collector can invoke the *FDExpungeAtomic* primitive,[34] which removes the directory entry, and expunges the file. If, on the other hand, the transaction has to be aborted, the *FDDelete* can be undone by the *FDUndeleteAtomic* primitive, since the file has not been expunged.

The *FDSync* operation ensures that any buffered directory information pertaining to the specified file is synchronized with the version saved on disk. If a file name is not specified, the contents of the entire directory buffer are synchronized.

The purpose of the *FDFind* primitive is to allow some of the convenience of hierarchical directories. In a file system with a hierarchical directory, it is usually possible to search for all files and subdirectories which exist in a particular directory. The *FDFind* primitive implements the same notion in our "flat" file name space. Given a file prefix (parallel to the full pathname of a directory) it can find all the components (filenames or subdirectory names contained in the directory). It is also possible to find the full tails of all matching file names (parallel to doing a recursive directory search). The boolean variable *comp* determines which type of search is undertaken: for matching components or tails. Since the buffer for returning the matching elements may not be able to hold the entire list of matches found, the last matching element is returned as the result of the *FDFind* primitive. The search can begin after this element when the next *FDFind* operation is invoked.

Three primitives are provided for manipulating the directory entries. The *FDInsert* primitive allows a new entry to be inserted in the directory. The *FDGet* primitive allows the directory entry for a particular file to be read, and the *FDRemove* primitive removes the entry for a particular file. These primitives can be used by the Client Arobject Interface to provide several useful functions to the client. For example, the *RenameFile* primitive is implemented by first obtaining file information using *FDGet*, then removing the old directory entry with *FDRemove*, and finally adding a new directory entry with *FDInsert* to correspond to the new file name.

The *FDRestart* primitive is primarily an initialization operation, invoked when the FD arobject is created. A cleanup operation on the logical disk associated with this FD arobject instance is initiated. The directory is checked for all normal files, and the Page Set Subsystem is asked to free all page sets corresponding to these files. Their directory entries are also removed. The *FDUnmount* operation is invoked when the logical disk associated with this FDM instance has to be unmounted cleanly. Once an *FDUnmount* has been received, FDM does not accept any new operations except *FDClose*. When all currently open files have been closed, it sends a reply to the requestor of *FDUnmount*, and destroys itself.

---

[34] The *FDExpungeAtomic* operation can alternatively be invoked by the transaction manager, or even the client.

6.6.4.1 Components of the Directory Management Subsystem

The FDM Subsystem consists of three main components: (1) the Directory Manager process, (2) the buffer for the directory B-tree, and (3) the Open Files Table. In addition to these components, a few other items of information are also maintained, such as the logical disk ID, and some flags. The three components are shown in Figure 6-18, and described briefly in this section.

The Directory Manager process is responsible for providing all of the functionality of the FDM subsystem. It accepts all requests for FDM operations, and returns the results. It manages all the FDM data structures as well. In order to maintain the directory buffer, it invokes operations on the Page Set Subsystem.

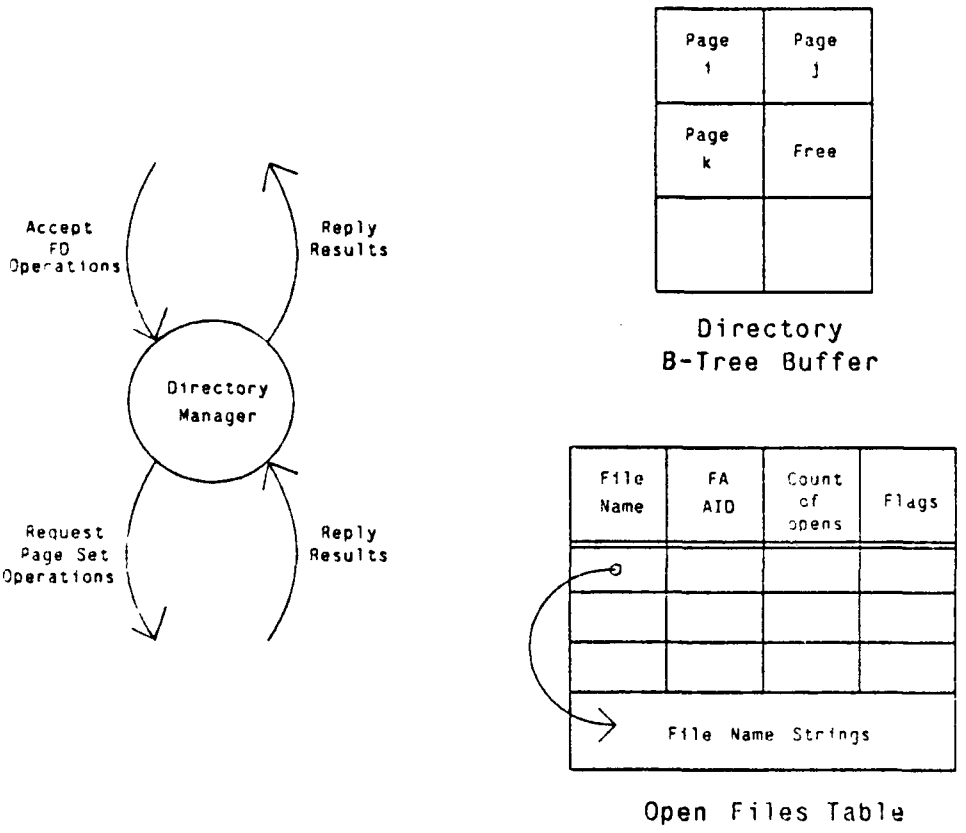


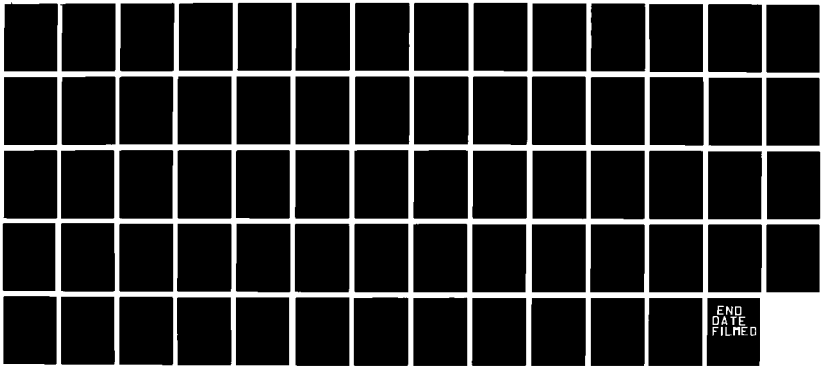
Figure 6-18: Components of the Directory Management Subsystem

The file system directory uses a B-tree structure, and is saved on disk as a number of [illegible] sets (refer Section 6.6.4.2 for details). Some of the pages of the directory [illegible]

AD-A184 491

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
CONCEPTS & TECHNIQUES FOR SUPPORT OF REAL TIME
DISTRIBUTED OPERATING SYSTEMS
BY: CARNEGIE-MELLON UNIVERSITY

3 OF 3
NOSC TD 1113
UNCLASSIFIED
JUL 87



memory by the FDM subsystem, in a data structure known as the Directory Buffer. The FDM Manager process interacts with the Page Set Subsystem in obtaining directory pages from disk, and in writing back any dirty buffered pages.

The Open Files Table keeps track of the files which are open. It maintains the names of these files, their arobject IDs, and the number of outstanding opens on each file. In addition, it maintains some flags, such as for open files which are deleted (or expunged). Whenever an *FDOpen* request is received, the OFT is first checked to see whether the file has already been opened or not. When a file is opened for the first time, an entry is made in the OFT, but for subsequent opens, the count of outstanding opens is incremented. For each *FDClose* operation, the count of outstanding opens for that file is decremented. If the count becomes zero, the entry is removed from the OFT, and the file arobject is told to deactivate itself. If an *FDDelete* operation is received on an open file, a DELETE flag is set in the OFT. Further *FDOpen* operations are not allowed, but *FDClose* operations are accepted, to achieve a clean deletion. Once a DELETE flag has been set, the file arobject is told to kill itself at the time of the last close operation.[35]

6.6.4.2 The Directory B-Tree

The file system directory maps file names to global page set IDs. It also saves some static information for each file: the date and time of creation and last modification, the file type, the file length, and some flags (e.g. the DELETED flag, to mark files which have been deleted but not expunged). Logically, the file system directory fragment on a particular disk is an independent B-tree. The entries in this B-tree are ordered by the filename (in alphabetical order). Each node of the directory B-tree is implemented as a page set. Thus, the directory is implemented as a set of page sets, which are maintained on disk by the Page Set Subsystem.

The structure of a node in the directory B-tree is somewhat complex (refer Figure 6-19). Each node of the directory B-tree contains a large number of entries (200 to 250). Logically, each directory entry is a triple consisting of the filename, the global page set ID, and a pointer to the file information block. The actual implementation is complicated by the fact that file names have variable sizes. Hence, instead of keeping the file name as the first element of the triple, a pointer to the file name is kept. This file name pointer is a byte offset from the start of the area for saving the full file names.

Each node of the directory B-tree is a page set. The first page of this page set is the header page,

---

[35] If an atomic file is deleted in this way, its file arobject is deactivated, and not killed. A flag is set in the directory entry for the file, indicating that it has been deleted, and further operations on that file, other than FDUndeleteAtomic or FDExpungeAtomic, are not allowed.

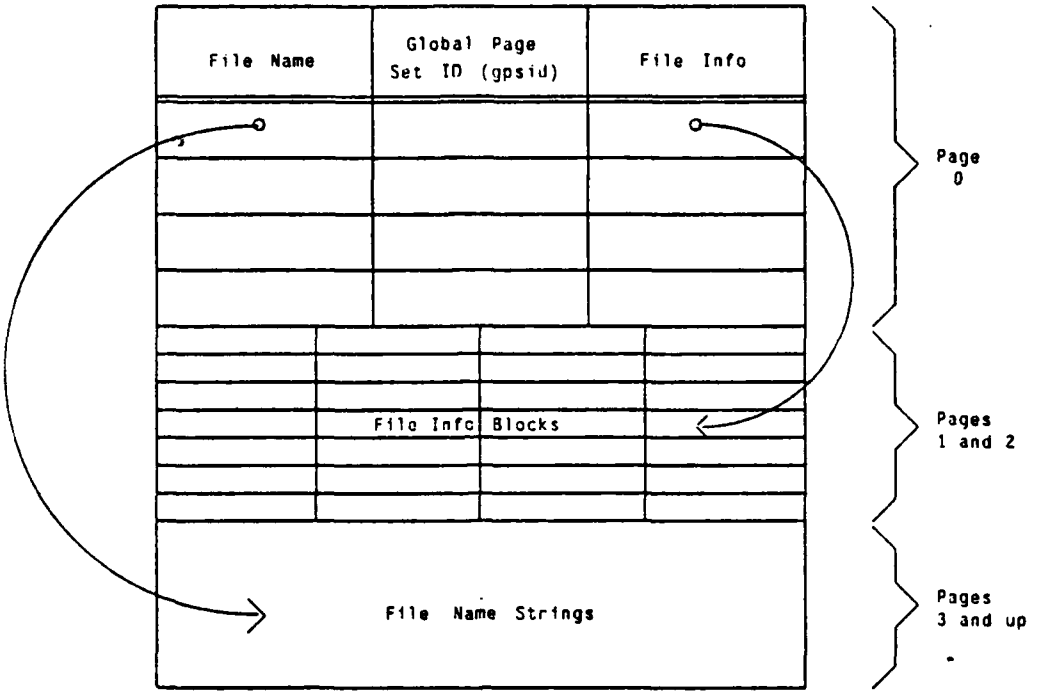


Figure 6-19: Structure of a Directory B-tree Node

which contains a series of triples. The first element of the triple is a pointer to the full file name, the second is the global page set identifier, and the third is a pointer to the information block (fixed size) for this file. If the node is internal (not a leaf node in the tree), the page set ID points to the node of the B-tree to be searched next. However, if the node is external (a leaf node), the page set ID refers to the location of the file (ID of the root page set for the file). The second and third pages of the node page set are reserved for the file information blocks. These pages will be unused in the case of internal nodes. The full file names are stored starting at the fourth page. Hence, the pointer to the file name holds a byte offset from the start of the fourth page.[36]

[36] For efficiency reasons, the file name pointer may actually consist of two parts: a *prefix* pointer, and a last *component* pointer. Since file names are typically long, and many of the file names for a given directory node differ only in the values of their last components, this scheme of storing names should significantly reduce directory storage requirements

6.6.4.3 FDM File Manipulation Operations

In this section, we will briefly discuss the key interactions of the of FDM file manipulation operations: *FDOpen*, *FDClose*, *FDCreate*, *FDDelete*, *FDUndeleteAtomic*, and *FDExpungeAtomic*. Each of these operations (except *FDClose*) is invoked by some Client Arobject Interface, and the result is returned there.

When the operation *FDOpen* is requested, the Open Files Table is first checked to see whether the file has already been opened. If this is indeed the case, the arobject ID for the file can be determined from the OFT. No further action is required, except the updating of the count of opens for the file. However, if this is the first open request, the inactive file arobject has to be activated. First the directory B-tree is searched (by bringing appropriate directory pages into the Directory Buffer) to find the global page set ID (GPSID) for the (root page set of) file in question. Then the Arobject/Process Management Subsystem is called to activate the arobject for the given page set, and return the file arobject ID. An entry for this file arobject is now made in the OFT. Also, the newly created file arobject is initialized by invoking the *FARestart* primitive. This completes the *FDOpen* sequence, and the new file arobject ID is returned to the FDM subsystem.

The *FDCreate* sequence is similar in flavor to the *FDOpen* sequence. First the directory B-tree is checked to be sure that the file does not already exist. Next, the Arobject/Process Management Subsystem is invoked to create a new file arobject of the specified type. The A/PM also decides where the new file is to be placed[37]. The GPSID and the arobject ID for the newly created file arobject are returned by A/PM. An entry for the file is created in the directory, and in the OFT. Finally, the new file arobject is "restarted" (with *FARestart*), and its ID is returned to the FDM subsystem.

The *FDClose* primitive is always invoked by the file arobject, when a client has requested an *FAClose* operation. The FDM subsystem responds to this operation by specifying one of three actions for the file arobject: CONTINUE, DEACTIVATE, or KILL. When *FDClose* is requested, first the count of outstanding open requests in the OFT is decremented. If some outstanding opens still remain, the CONTINUE command is given to the file arobject. If however, there are no outstanding opens, the entry for this file in the OFT is deleted. If the file has been modified (*modified* is TRUE), the information blocks for the file are updated. Finally, the file arobject is asked to deactivate itself. The last *FDClose* operation on a file is somewhat different if the file has already been flagged for deletion by some other client. The directory entry for the file is removed, and the file arobject is asked to kill itself (which automatically removes the associated page sets).

[37] The file placement algorithm is subsumed by the arobject and process placement algorithms, and is implemented by the Arobject/Process Management Subsystem. If a preference for placement is specified, this is taken into account in making the final decision.

The implementation of the *FDDelete* primitive is quite straightforward. If the file arobject is inactive when the primitive is invoked, the Arobject/Process Management Subsystem is called to destroy the inactive file arobject (which removes the relevant page sets), and the directory entry for the file is removed. If the file is active at the time *FDDelete* is called, the delete flag is set for the file in the OFT. At the time of the last *FDClose* operation, the file is deleted by destroying the active file arobject.

The handling of the primitives (described above) for atomic files may be somewhat different. Creating, deleting, opening, and closing files atomically is a part of the bigger and more general problem of creating, deleting, activating, and deactivating arobjects atomically. Hence, these issues have to be addressed in a unified manner. At present, some hooks have been provided, which help in treating these primitives (especially *FDDelete*) as compound transactions with compensation actions. Thus, when an atomic file is to be deleted, the directory is merely flagged as deleted, but the arobject is not destroyed. Once the highest level transaction has committed, the garbage collector can expunge all "deleted" files, by calling *FDExpungeAtomic*. If the transaction has to abort, the deleted file can be reclaimed by calling *FDUndeleteAtomic*.

6.6.5 File Management Scenarios

In this section, we will show how the four subsystems within the File Subsystem interact with one another, and with other subsystems in the kernel, to provide the specified functionality. We will examine the operations most frequently encountered in the lifetime of a file, and thereby explain the interactions within the File Subsystem.

The *CreateFile* primitive is invoked by the client when a new file has to be created. In response to this request, the Client Arobject Interface has to first determine the ID of the Directory Management arobject which will manage the directory fragment corresponding to the new file. The Client Arobject Interface first invokes the *FPMap* primitive (refer Figure 6-20). The *FPMap* operation is accepted by the FPMM Manager process, which checks the prefix of the filename in the Prefix Map Table, to determine the ID of the corresponding FDM arobject (let this value be *fdm-aid*). The result of the *FPMap* operation (*fdm-aid*) is returned to the Client Arobject Interface. The CA Interface now invokes the *FDCreate* operation on the correct instance of the Directory Management Arobject (*fdm-aid*). The FDM Manager reads relevant pages of the directory into the buffer, and ensures that the file does not already exist. It then calls the Arobject/Process Management Subsystem to create a file arobject (using the *CreateArobject* primitive). The Arobject/Process Management Subsystem returns the ID of the newly created file arobject. Next, the *FetchArobjectStatus* primitive is used to determine the global page set ID for the root page set corresponding to this arobject. The Directory Manager then

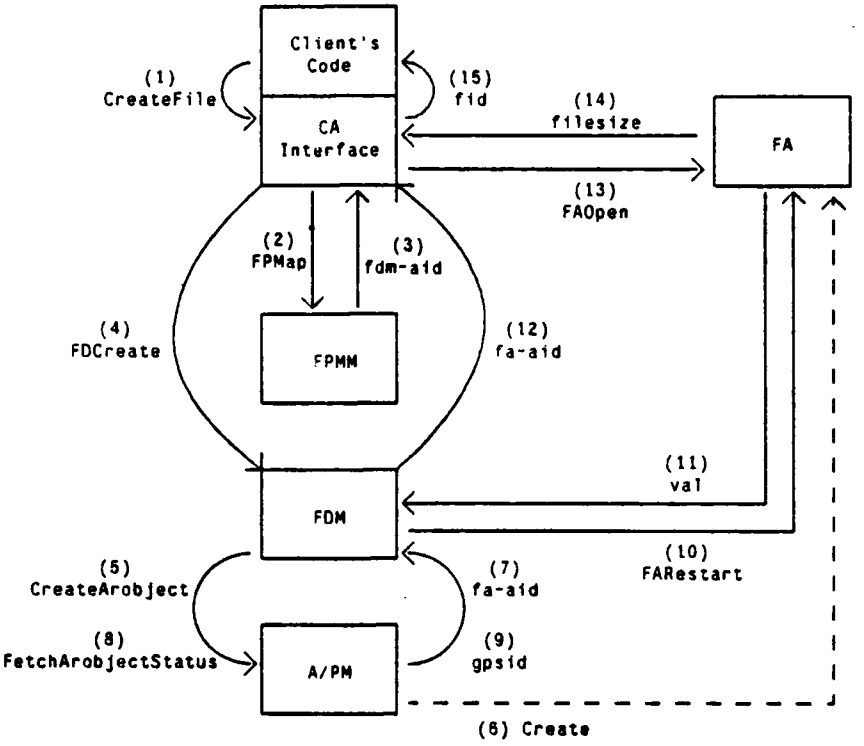


Figure 6-20: Interactions in the Implementation of the CreateFile Primitive

makes an entry in the directory for the new file, showing the mapping of its filename and its GPSID. It writes the creation date in the file information block. It also invokes the *FARestart* operation on the newly created file arobject to initialize it, and provide some information such as the file size. The FDM Manager makes an entry for the file in the Open Files Table, and returns the ID of the file arobject to the Client Arobject Interface. The CA Interface calls the File Arobject with an *FAOpen* request. The open operation is carried out, and the current filesize is returned. The CA Interface then returns to the client with the ID of the file.[38]

The interactions of the *OpenFile* primitive are quite similar to those of the *CreateFile* primitive. First, the *FPMap* primitive is invoked by the Client Arobject Interface. Then, the *FDOpen* operation is invoked. If the file has not already been opened by some other client, the FDM subsystem reads the directory into the buffer to determine the global page set ID for the file. It then invokes the Arobject/Process Management Subsystem to activate the inactive file arobject corresponding to the

[38] The *file ID* returned to the client arobject is different from the ID of the corresponding file arobject. The CA Interface maintains a mapping between the longer file arobject ID, and the shorter file ID which it generates.

GPSID. The active file arobject is initialized by *FARestart*, and an entry is made in the Open Files Table. The file arobject ID is returned to the CA Interface subsystem. Finally, *FAOpen* is called, and the interactions of *OpenFile* are complete.

The interactions in *ReadFile* and *WriteFile* are quite straightforward. The *ReadFile* and *WriteFile* operations given by the client are buffered by the CA Interface. The *FARead* and *FAWrite* operations are invoked by the CA Interface as necessary. This reduces the number of interactions with the file arobject.

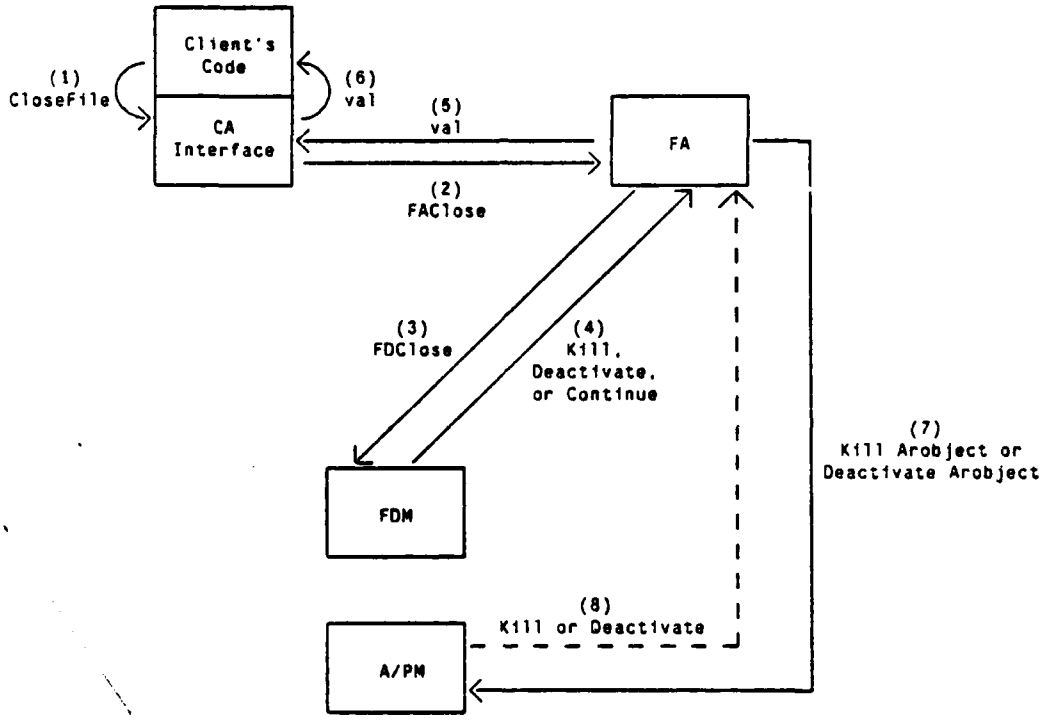


Figure 6-21: Interactions in the Implementation of the CloseFile Primitive

The *CloseFile* operation given by the client is translated to *FAClose* by the CA Interface (refer Figure 6-21), after first flushing the CA Interface buffers. The file arobject removes the client from the Open Client List, and invokes the *FDClose* operation. The information block for the file is updated, and the file is closed as explained in Section 6.6.4. The file arobject is told to CONTINUE, DEACTIVATE, or KILL itself, depending on other outstanding open requests, and whether the file has been deleted by some other client. The file arobject replies to the *FAClose* operation, and then executes the specified

action. If it has to deactivate or kill itself, it calls on the Arobject/Process Management Subsystem to perform the necessary operation.

The interactions for *DeleteFile* are quite simple. The *DeleteFile* operation is translated to *FDDelete* by the CA Interface, after first using the *FPMap* operation if necessary. The FDM subsystem calls the Arobject/Process Management Subsystem to destroy the inactive file arobject, and then removes its directory entry. If the file is open at the time the operation is invoked, a DELETE flag is set in the Open Files Table, and the deletion is completed at the time of the last *FDClose* operation.

6.7 Page Set Subsystem

The main purpose of the Page Set Subsystem is to provide an abstract interface to the physical secondary memory storage devices (disks). The abstraction provided is that of a set of *logical disks*, each of which holds some number of *page sets*. A logical disk is a contiguous region (partition) of a physical disk, consisting of *N* logical pages, numbered from 0 to *N*-1.[39] Each physical disk can contain at most one ArchOS logical disk. However, the logical disk can be located anywhere on the physical disk, and can possibly cover the entire disk. Each logical disk has a unique identifier (logical disk ID) permanently associated with it, i.e. it will always have the same logical disk ID, regardless of which node or disk drive the disk is mounted on.

A page set is (conceptually) an infinite set of pages, numbered 0, 1, 2, ... Any pages of a page set which have not been explicitly written, are defined as containing all zeros.[40] Pages can be read or written in any order within a page set, i.e. the pages of a page set are randomly accessible, and sparse page sets are permitted. All of the pages comprising a single page set will be located on the same logical disk. This helps reduce the number of page sets which would be affected by any single disk or node failure. Every page set is given a page set ID, which is unique within the logical disk on which it resides. A globally unique page set identifier (GPSID) can then be constructed by combining the logical disk ID with the page set ID.

ArchOS supports two different classes of page sets: *standard page sets*, and *atomic page sets*. Standard page sets themselves come in three types: *temporary*, *permanent*, and *dual*. Temporary page sets are intended to provide short term storage for data on secondary memory. They are destroyed, and their pages automatically reclaimed by the system, following a crash. Temporary page

---

[39] Currently a page is defined to be 2K bytes in size, but that is a parameter which can be changed and tuned to obtain "optimum" performance.

[40] Of course, such unwritten pages do not occupy any space on the logical disk.

sets are primarily used as the paging areas for the "volatile" segments of process address spaces, such as the User Stack, User Heap, and Shared Normal Segments (see Section 6.2.3.5). In particular, normal file arobjects (see Section 6.6) store all of their data in volatile segments, and hence in temporary page sets.

Permanent page sets are primarily used to store *permanent* abstract data type instances, such as permanent files. Permanent page sets survive system crashes, but their consistency following a crash is not guaranteed. If a permanent page set was being actively modified at the time of a crash, some of the modifications may be permanently recorded while others are lost. Furthermore, if a page was being written precisely at the time of the crash, that page could be written incorrectly, resulting in the loss of some data. Note that permanent page sets closely resemble the notion of "files", as commonly found in more conventional operating systems.

Dual page sets are like permanent page sets, except that each page is written atomically. If a crash occurs while writing a page in a dual page set, either the new contents of the page will be permanently and correctly recorded, or the original contents will be recovered (as if the write operation had never occurred). The atomic writing of an individual page is accomplished by carefully writing *two* copies of the page (hence the name "dual" page set). For more details, see the *explanation of "stable storage" in [Sturgis 80]. Dual page sets are used for storing* key data structures, such as file system directories, for which it is important to minimize the amount of damage in the event of a crash.

The final type of page set supported by ArchOS is the atomic page set. Atomic page sets actually represent a second major class of page sets, differing (somewhat) from standard page sets, both in terms of their implementation and their user interface. Atomic page sets permit the grouping of operations into atomic transactions, (either all of the operations will be completed, or none of them will be). The operations comprising a single transaction can involve multiple atomic page sets, stored on any number of different logical disks, and spanning any number of system nodes. Furthermore, operations are not restricted to being page-oriented. Instead, arbitrary sequences of bytes can be read or written. Thus, two separate transactions can modify different parts of a single page, without interference or unexpected inconsistencies arising.

Atomic page sets also support the notion of nested transactions. Both elementary and compound transactions can be arbitrarily nested, as explained in Section TMSEC. The *commit* or *abort* of a (sub)transaction will be properly reflected in the contents of the affected atomic page sets. To support this, the atomic page sets contain the intermediate states of all of the incomplete, nested

transactions, in addition to the "current" page set contents. Atomic page sets provide complete failure atomicity for "soft and clean" failures, as defined in [Bernstein 83]. However, the consistency of atomic page sets in the face of "concurrent" updates from separate transactions is not automatically guaranteed. The lock management primitives of the Transaction Subsystem must be used to ensure such consistency (see Section TMSEC). Atomic page sets are used to store *atomic* abstract data type instances, such as atomic files.

The Page Set Subsystem is implemented using four different types of kernel arobjects. Each node of the distributed computer system will contain one or more instances of these four arobjects, as illustrated in Figure 6-22. For each physical disk that is mounted on a node, a corresponding instance of the Logical Disk Subsystem, the Standard Page Set Subsystem, and the Atomic Page Set Subsystem will be created. In addition, each node will contain a single instance of the Page Set Restart/Reconfiguration Subsystem. Figure 6-22 shows the "uses" relationships among these component subsystems.

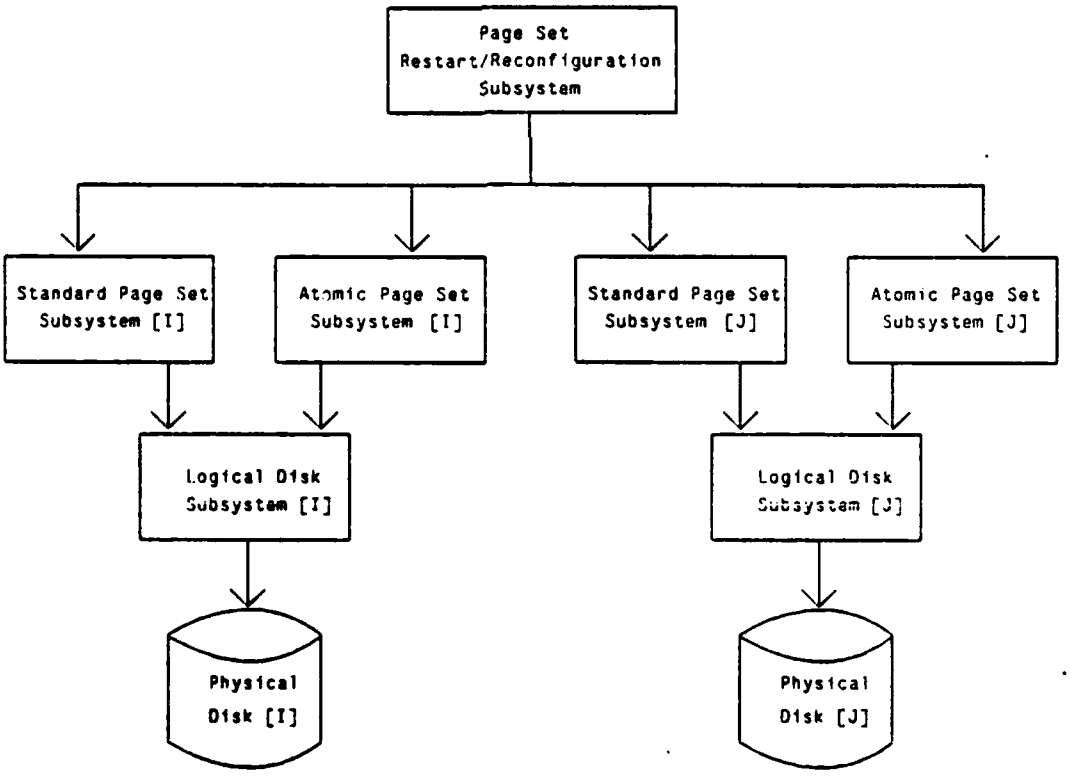


Figure 6-22: Major Components of the Page Set Subsystem

The Logical Disk Subsystem builds the logical disk abstraction (discussed above) on top of the available physical disk hardware. It manages the allocation of pages on the logical disk, and supports multi-page read/write operations to/from either local or remote primary memory (buffer) areas. The Standard Page Set Subsystem provides support for the three types of standard page sets: temporary, permanent, and dual. It constructs these page sets using the facilities of the corresponding Logical Disk Subsystem. Similarly, the Atomic Page Set Subsystem manages all of the atomic page sets that are stored on the corresponding logical disk. Finally, the Page Set Restart/Reconfiguration Subsystem (one per node) is responsible for constructing the three "per-disk" subsystems, for each disk that is mounted on its node. It allows disks to be "dynamically" added and removed from the system, without requiring the associated node to be completely restarted. In conjunction with this, the Page Set Restart/Reconfiguration Subsystem cooperates with its peers on other nodes, to maintain a global list (replicated at each node) of all the disks mounted anywhere within the distributed computer system.

6.7.1 Standard Page Set Subsystem

An instance of the Standard Page Set Subsystem kernel arobject is associated with each logical disk in the system. The Standard Page Set Subsystem supports three types of page sets: temporary, permanent, and dual. The same set of operations are provided for each of these types of page sets, but the semantics of some of the operations vary slightly, depending on the page set type. For example, the writing of pages in a dual page set is handled differently than in the case of a temporary or permanent page set. The Standard Page Set Subsystem supports multi-page read/write access to the contents of page sets, allowing greater efficiency than would be possible with a page-at-a-time interface. The following is the complete list of primitives provided by the Standard Page Set Subsystem:

---

psid = **PSCreate**(type)
val = **PSDestroy**(psid)

npr = **PSRead**(psid, pnum, npages, buffer)
val = **PSIsZero**(psid, pnum, npages)
npw = **PSWrite**(psid, pnum, npages, buffer)
npw = **PSZero**(psid, pnum, npages)
val = **PSMove**(psid, pnum, npages, to-pnum)

val = **PSSync**([psid])
val = **PSStatus**(psid, statusbuffer)
val = **PSRestart**(disk-aid [, fast])

PSID psid — The identifier for the page set (unique within the logical disk). It includes an indication of the page set type: TEMPORARY, PERMANENT, or DUAL.

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

INT npr, npw — The actual number of pages read or written.

PSTYPE type — The type of page set to be created: TEMPORARY, PERMANENT, or DUAL.

INT pnum — The starting page number, within a page set, for the specified operation.

INT npages — The number of pages involved in the specified operation.

BUFFER *buffer — The buffer address in the kernel address space of either the local or remote node.

INT to-pnum — The destination page number, to which the specified pages are to be moved.

PSSTATUSBUFFER *statusbuffer — The buffer address for returning status information (can be either a local or remote kernel address).

AID disk-aid — The arobject ID of the Logical Disk Subsystem, associated with the disk to be used by this instance of the Standard Page Set Subsystem.

BOOLEAN fast — TRUE if this is to be a fast restart, avoiding all of the disk checking and garbage collection; otherwise FALSE. The default is FALSE.

The *PSCreate* primitive is used to create a new page set, of the specified *type*, on the disk associated with this instance of the Standard Page Set Subsystem. The identifier for the new page set (*psid*), which is unique within this logical disk, is returned. Initially, no pages are actually allocated to the page set, i.e. reading any page will return all zeros. The *PSDestroy* primitive frees all pages belonging to the specified page set (*psid*), and removes all record of that page set from the disk.

PSRead reads multiple (*npages*) pages, beginning with *pnum*, from page set *psid*, into the given primary memory *buffer*. If the buffer is located on a remote node, the required copying of data across the network will be handled automatically by the Logical Disk Subsystem (see Section 6.7.3). Any pages which have never been explicitly written will be read as all zeros. *PSRead* returns the number of (non-zero) pages which were actually read. *PSIsZero* can be used to check whether the specified range of pages, in the given page set, are all zero (unallocated). If so, it returns TRUE; otherwise FALSE.

PSWrite writes multiple (*npages*) pages from the given primary memory *buffer*, into page set *psid*, beginning at page number *pnum*. The number of pages actually written is returned. As with *PSRead*, the *buffer* can be located on either the local or a remote node. Any necessary cross-network data copying will be handled automatically by the Logical Disk Subsystem. *PSZero* provides a means for efficiently "zeroing" (deallocating) a range of pages within a page set. It is especially useful for "truncating the tails" of page sets, which is accomplished by zeroing the highest numbered (allocated) pages. The number of pages actually zeroed (i.e. the number of previously allocated pages which have now been deallocated) is returned.

PSMove can be used to "move" the specified range of pages within page set *psid*, to the new location specified by *to-pnum*. Overlapping source and destination ranges are permitted, with the result being equivalent to first deallocating all of the pages in the source and destination regions, followed by rewriting the original source pages into their destination locations. Among other things, *PSMove* can be used for "truncating the heads" of page sets, by specifying page zero as the destination, and moving all of the tail pages forward.

The *PSSync* primitive ensures that any buffered information within the Standard Page Set Subsystem is consistent (synchronized) with the information on secondary memory. If a particular page set (*psid*) is specified, only buffered information related to that page set is guaranteed to be consistent with the secondary memory information.[41] The *PSStatus* primitive returns (in *statusbuffer*) information about the specified page set (*psid*). This includes the page set type, its maximum allocated page number (virtual size), and its actual number of allocated pages (physical size).

PSRestart initializes the Standard Page Set Subsystem, for the logical disk specified by *disk-aid*. It checks to ensure that the main secondary memory data structure (the Standard Page Set B-Tree, discussed below) is accessible and consistent, and ensures that the pages of any *dual* page sets are also consistent, by using the *DKRecoverDual* primitive of the Logical Disk Subsystem (see Section 6.7.3). A side effect of using *DKRecoverDual* is that all of the pages in the Standard Page Set B-Tree, and all of the dual page set data pages, will be marked as allocated. *PSRestart* then marks (as allocated) all of the data pages belonging to any *permanent* page sets stored on the disk (see the discussion of *DKMark* in Section 6.7.3). Finally, *PSRestart* "destroys" any *temporary* page sets which still reside on the disk, by carefully removing their entries from the Standard Page Set B-Tree.

---

[41] This may require much less time than synchronizing *all* of the buffered information.

[42]*PSRestart* should only be invoked when the Standard Page Set Subsystem is first created by the Page Set Restart/Reconfiguration Subsystem (see Section 6.7.4 below), i.e. when the associated logical disk is first added to the system. The optional parameter (*fast*) can be used to indicate that a fast restart is in progress, and hence all of the consistency checks, page allocation marking, and removal of temporary page sets can be skipped. For more details concerning restart and garbage collection activities, see Section 6.7.4.

6.7.1.1 Components of the Standard Page Set Subsystem

Each instance of the Standard Page Set Subsystem kernel arobject (one per logical disk) has the simple structure illustrated in Figure 6-23. It consists of two main components: a single Manager process, and a B-Tree Buffer. The Manager handles all of the Standard Page Set primitives, discussed above. It uses the facilities of the associated Logical Disk Subsystem, in order to store, access, and manipulate the page sets recorded on its corresponding logical disk. Note that the Standard Page Set Subsystem itself does no buffering of data pages. It is left to the Logical Disk Subsystem to transfer data directly into and out of client buffers.

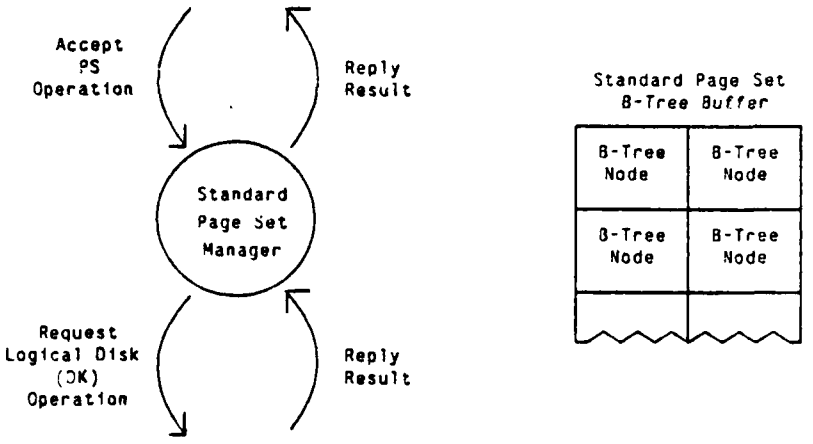


Figure 6-23: Components of the Standard Page Set Subsystem

[42]It is assumed (unless a "fast" restart is in progress) that all of the pages on the logical disk have been "freed" prior to invoking *PSRestart*. As a result, the page allocation information must be reconstructed, and until it is, all requests to allocate new logical disk pages must be avoided. *PSRestart* must mark (as allocated) all of the pages which are part of the Standard Page Set Subsystem. Since none of the data pages from *temporary* page sets are marked, they are automatically freed. However, the record of the pages allocated to temporary page sets is still stored in the Standard Page Set B-Tree, and must be removed. The removal of entries from the B-Tree is a "safe" operation, since it does not require the allocation of any new logical disk pages, although it may involve the freeing of some pages from the B-Tree.

The B-Tree Buffer holds the most recently accessed nodes (pages) of the Standard Page Set B-Tree for this logical disk. The Standard Page Set B-Tree is the data structure used to record the sets of logical disk pages, comprising the standard page sets stored on this disk. The use of a B-tree [Comer 79] for this purpose is quite similar to its use in the Xerox Distributed File System (XDFS) [Sturgis 80, Mitchell 82]. The logical structure of the Standard Page Set B-Tree is illustrated in Figure 6-24.

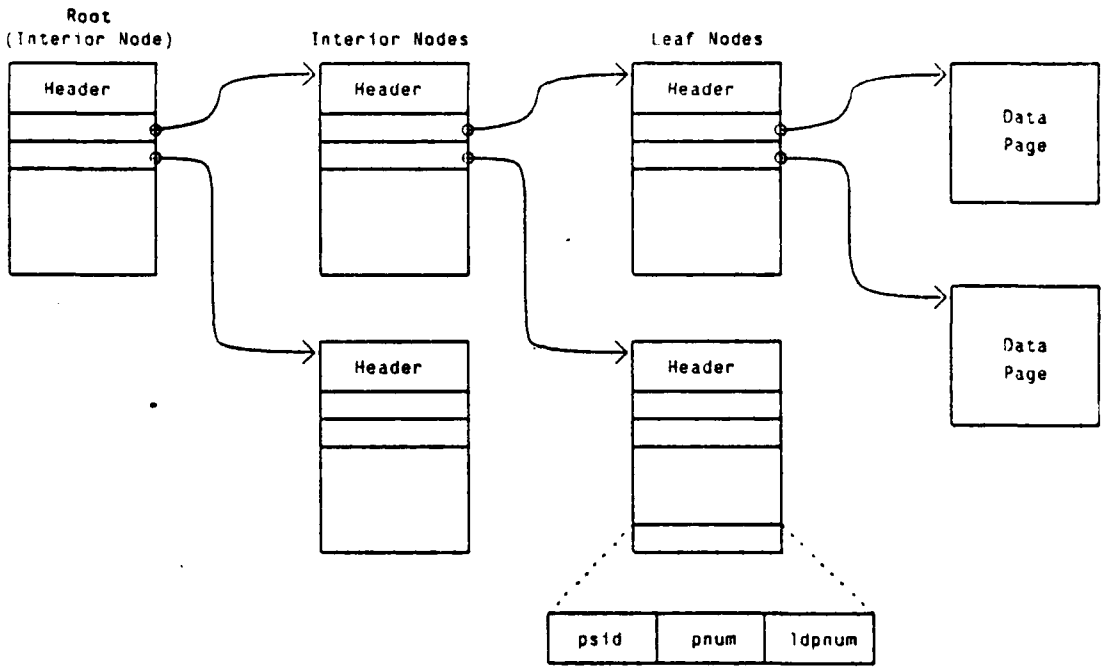


Figure 6-24: Standard Page Set B-Tree

In essence, the B-tree maintains a sorted list of data pages, ordered by page set ID (*psid*), and page number within *psid*. This allows the mapping from (*psid*, *pnum*) to the logical disk page number (ldpnum) to be be performed very quickly. Each node of the B-tree is stored as a *dual* logical disk page, in order to improve the reliability of the data structure. In addition to page mapping information, each node contains a small amount of header information, which indicates the type of node (interior or leaf), and the number of page map entries. Every map entry, whether in an interior or a leaf node, has the same structure (*psid*, *pnum*, *ldpnum*). However, the type of page pointed to by *ldpnum* will differ, depending on whether the node is interior or a leaf. A pointer to the root node of the B-tree (its *ldpnum*) is stored in page zero of the logical disk, so that the Standard Page Set Subsystem can

always find its B-tree data structure.[43]

Note that the type of each page set (temporary, permanent, or dual) is included in the page set ID (*psid*). When a page set is first created, an entry for page number zero is added to the B-tree, with *ldpnum* set to zero to indicate that the page has not really been allocated yet. As pages are written or zeroed, appropriate entries are added or removed from the B-tree. However, there will always be an entry (whether allocated or not) for page zero of each page set that exists. Note that the structure of the B-tree allows all of the status information for each page set (its type, logical size, and physical size) to be determined quite easily. It also permits the pages of a page set to be written, read, and zeroed very efficiently.

In terms of storage overhead, the B-tree structure is also quite reasonable. If we assume that *psid*, *pnum*, and *ldpnum* are each 32 bits (4 bytes) long, then a single B-tree node (2K byte page) can contain 170 map entries, with 8 bytes remaining for header information. Since each leaf page of the B-tree is *dual*, and can map 170 data pages, only a little over one percent of the logical disk pages belonging to the Standard Page Set Subsystem should be needed for storing the B-tree itself. Furthermore, it should be noted that with 170 entries per node, a three level B-tree can map 170^3 data pages, which is considerably more than can be contained on any disk that is likely to be used as an ArchOS logical disk. Thus, the Standard Page Set B-Tree will be a maximum of three levels deep.

A special note should be made concerning the Standard Page Set B-Tree Buffer, illustrated earlier in Figure 6-23. Although (for clarity of exposition) each instance of the Standard Page Set Subsystem is shown as having its own buffer area, in practice, all instances on a single node would share a common Kernel Page Buffer Pool. Indeed, the Kernel Page Buffer Pool would be shared with many other subsystems on that node as well: Logical Disk Subsystems, Atomic Page Set Subsystems, File System Disk Directory Management Subsystems, and so on. Each subsystem allocates pages from the Kernel Page Buffer Pool (in least recently used order) as required. Pages are then returned to the pool (freed) as soon as they are no longer being actively used. Each subsystem maintains a list of the "free" pages in the buffer pool, which it has used recently. This list provides "hints" about pages in the buffer pool which may still contain information of use to the subsystem, i.e. they are buffer pages which can be reclaimed, rather than reading the information again from disk, assuming the pages have not already been reused by some other subsystem.

---

[43] Page zero of each logical disk is a special *dual* page, which contains pointers to all of the major data structures on the disk. It also contains information about the disk itself, such as its size and its logical disk ID.

6.7.2 Atomic Page Set Subsystem

The Atomic Page Set Subsystem is quite similar to the Standard Page Set Subsystem, except that it provides the "atomic page set" abstraction. The primary difference between the standard and atomic page set abstractions is that the consistency of atomic page sets will be maintained in the event of a crash. In addition, the Atomic Page Set Subsystem allows arbitrary sequences of bytes to be read or written, rather than restricting the operations to being page-oriented. Atomic page sets are primarily used for storing *atomic* abstract data type instances, such as atomic files. As with the Standard Page Set Subsystem, an instance of the Atomic Page Set Subsystem is created for each logical disk in the distributed computer system.

The Atomic Page Set Subsystem provides primitives for "committing" arbitrarily nested elementary and compound transactions, in two phases: the prepare phase, and the commit phase. These two phases can be used by the Transaction Management Subsystem, as part of its three phase commit protocol (see Section TMSEC). Using the mechanisms provided here, it is possible to atomically commit (or to abort) transactions which span multiple logical disks, and multiple nodes. It should be noted that the Atomic Page Set Subsystem only provides support for the properties of *failure atomicity*, and *durability* [Eswaran76, Gray81]. The sequencing and scheduling of concurrent transactions, so as to ensure *consistency*, is assumed to be handled by the Transaction Management Subsystem. The actual primitives provided by the Atomic Page Set Subsystem are the following:

---

psid = **APSCreate**(tid)
val = **APSDestroy**(tid, psid)

nbr = **APSRead**(tid, psid, location, nbytes, buffer)
val = **APSIsZero**(tid, psid, location, nbytes)
nbw = **APSWrite**(tid, psid, location, nbytes, buffer)
nbw = **APSZero**(tid, psid, location, nbytes)
val = **APSMove**(tid, psid, location, nbytes, to-location)

val = **APSPrepareCommit**(tid)
val = **APSCommit**(tid)
val = **APSAbort**(tid)

val = **APSStatus**(psid, statusbuffer)
val = **APSRestart**(disk-aid [, fast])

PSID psid — The identifier for the atomic page set (unique within the logical disk). It includes an indication of the page set type (ATOMIC).

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

INT nbr, nbw — The actual number of bytes read or written.

TID tid — The ID of the transaction to which this operation belongs. From the *tid* it is possible to determine the transaction type, as well as the parent transaction ID (if this is a nested transaction).

INT location — The starting location (byte offset from the beginning of the atomic page set) for the specified operation.

INT nbytes — The number of bytes involved in the specified operation.

BUFFER *buffer — The buffer address in the kernel address space of either the local or remote node.

INT to-location — The destination location, to which the specified bytes are to be moved.

APSSTATUSBUFFER *statusbuffer — The buffer address for returning status information (can be either a local or remote kernel address).

AID disk-aid — The arobject ID of the Logical Disk Subsystem, associated with the disk to be used by this instance of the Atomic Page Set Subsystem.

BOOLEAN fast — TRUE if this is to be a fast restart, avoiding all of the disk checking and garbage collection; otherwise FALSE. The default is FALSE.

The *APSCreate* primitive is used to create a new atomic page set, on the disk associated with this instance of the Atomic Page Set Subsystem. A transaction ID (*tid*) is specified, so that the new atomic page set will only come into permanent existence when transaction *tid* commits.[44] Until then, the new page set is only visible within this transaction, or within any nested subtransactions of *tid*. *APSCreate* returns the identifier for the new atomic page set (*psid*), which is unique within this logical disk. Initially, no pages are actually allocated to the page set, i.e. reading any part of it will return all zeros. The *APSDestroy* primitive frees all pages belonging to the specified atomic page set (*psid*), and removes all record of that page set from the disk. As was the case with *APSCreate*, the specified transaction ID (*tid*) determines when the effects of *APSDestroy* will be permanently committed, as well as the scope of their visibility in the interim.

APSRead reads *nbytes* bytes into the given *buffer*, beginning from byte *location* in atomic page set *psid*. The transaction ID (*tid*) is taken into account when determining which modifications (if any) to

---

[44] If *tid* is a nested elementary transaction, the new atomic page set will only come into permanent existence when the top-level parent of *tid* commits.

the specified bytes, by transactions which have not yet committed, should be visible to this read operation. If the buffer is located on a remote node, the data will be automatically copied across the network, using the special kernel *Copy* primitive. Any bytes which have never been explicitly written will be read as zeros. *APSRead* returns the number of (non-zero) bytes which were actually read. *APSIsZero* can be used to check whether the specified range of bytes, in the given atomic page set, are all zero. If so, it returns TRUE; otherwise FALSE. The specified transaction ID serves the same purpose as in the case of *APSRead*.

APSWrite writes *nbytes* bytes from the given *buffer*, into atomic page set *psid*, beginning at byte *location*. The number of bytes actually written is returned. The specified transaction ID (*tid*) determines when the effects of *APSWrite* will be permanently committed, as well as the scope of their visibility in the interim. As with *APSRead*, the *buffer* can be located on either the local or a remote node. Any necessary cross-network data copying will be handled automatically, using the special kernel *Copy* primitive. *APSZero* provides a means for efficiently "zeroing" a range of bytes within an atomic page set. The specified transaction ID serves the same purpose as in the case of *APSWrite*. When complete pages are zeroed, they are removed from the page set (deallocated). *APSZero* is especially useful for "truncating the tails" of atomic page sets, which is accomplished by zeroing the highest numbered (non-zero) bytes. The number of bytes actually zeroed (i.e. the number of previously non-zero bytes which have now been zeroed) is returned.

APSMove can be used to "move" the specified range of bytes within atomic page set *psid*, to the new location specified by *to-location*. The transaction ID (*tid*) serves the same purpose as in the case of *APSWrite* and *APSZero*. Overlapping source and destination ranges are permitted, with the result being equivalent to first zeroing all of the bytes in the source and destination regions, followed by rewriting the original source bytes into their destination locations. Among other things, *APSMove* can be used for "truncating the heads" of atomic page sets, by specifying location zero as the destination, and moving all of the tail bytes forward.

The *APSPrepareCommit* primitive is used in phase one of the two phase transaction commit sequence for atomic page sets. All modifications to atomic page sets on this logical disk, which were made in the course of the specified transaction (*tid*), are carefully recorded in a special "Commit List" on secondary memory. These modifications can no longer be lost in the event of a crash. Transaction *tid* is flagged (on the disk) as "prepared to commit". The invoker is then informed (through the return value, *val*), of the Atomic Page Set Subsystem's readiness to commit the transaction. Following successful completion of the *APSPrepareCommit* primitive, the only allowed operation involving transaction *tid* is *APSCommit* or *APSAbort*. Note that *APSPrepareCommit* need

not be invoked in the case of a nested *elementary* subtransaction (it has no effect). Such a subtransaction does not actually modify the permanent contents of the atomic page sets, until its top level parent transaction commits. Hence, a two phase commit sequence is unnecessary, since any failure will cause the parent transaction, as well as this "committed" subtransaction, to be aborted.

APSCommit is used in phase two of the two phase transaction commit sequence. It is also the only primitive required to "commit" a nested elementary subtransaction. In this latter case, *APSCommit* does not actually modify the permanent contents of the atomic page sets. Instead, it simply makes all of the modifications which were made in the course of the specified transaction (*tid*) visible to the parent of *tid*. When *APSCommit* is applied to a top level elementary transaction, or to any compound transaction, the specified transaction (*tid*) must already have been "prepared" (with *APSPrepareCommit*), in phase one of the commit sequence. In this case *APSCommit* is the second (final) phase of the commit sequence, and it is responsible for carefully updating the "permanent" contents of the atomic page sets, based on the modification information contained in the Commit List. *APSCommit* first flags (on the disk) transaction *tid* as "committed". It then returns the result (*val*) to the invoker, allowing the invoker to continue its execution while the Atomic Page Set Subsystem makes the required modifications to the atomic page sets.

The *APSAbort* primitive is used to abort the specified transaction (*tid*). This involves removing (undoing) all modifications made within transaction *tid*, or any of its nested subtransactions (except already committed compound subtransactions). Note that transactions which have been "prepared", but not yet committed, can still be aborted. In that case the aborted modifications must be carefully removed from the Commit List.

The *APSStatus* primitive returns (in *statusbuffer*) information about the specified atomic page set (*psid*). This includes its virtual size (maximum non-zero byte location), physical size (actual number of allocated bytes), and an indication of which uncommitted transactions, if any, are still operating on this page set. Note that since no transaction ID is specified with the *APSStatus* primitive, the sizes reported are those of the current, "permanent" version of the atomic page set.

APSRestart initializes the Atomic Page Set Subsystem, for the logical disk specified by *disk-aid*. It checks to ensure that the main secondary memory data structures (the Atomic Page Set Permanent and Commit B-Trees, discussed below) are accessible and consistent, by using the *DKRecoverDual* primitive of the Logical Disk Subsystem (see Section 6.7.3). A side effect of using *DKRecoverDual* is that all of the pages in the Permanent and Commit B-Trees will be marked as allocated. *APSRestart* then marks (as allocated) all of the data pages belonging to any *atomic* page sets stored on the disk,

and also marks all of the data pages associated with the Commit List (see the discussion of *DKMark* in Section 6.7.3).[45] Finally, *APSRestart* completes the processing of any transaction which is in the Commit List, and flagged as "committed". This involves carefully updating the permanent contents of the atomic page sets, based on the modification information contained in the Commit List.[46] *APSRestart* should only be invoked when the Atomic Page Set Subsystem is first created by the Page Set Restart/Reconfiguration Subsystem (see Section 6.7.4 below), i.e. when the associated logical disk is first added to the system. The optional parameter (*fast*) can be used to indicate that a fast restart is in progress, and hence all of the consistency checks and page allocation marking can be skipped. However, any remaining commit processing must still be performed, even in the case of a fast restart. For more details concerning restart and garbage collection activities, see Section 6.7.4.

6.7.2.1 Components of the Atomic Page Set Subsystem

Each instance of the Atomic Page Set Subsystem kernel arobject (one per logical disk) has the general structure illustrated in Figure 6-25. It consists of five main components: a single Manager process, three B-Tree Buffers, and a Data Page Buffer. The Manager handles all of the Atomic Page Set primitives, discussed above. It uses the facilities of the associated Logical Disk Subsystem, in order to store, access, and manipulate the page sets recorded on its corresponding logical disk.

The four buffers in Figure 6-25 are primarily shown for clarity of exposition. In practice (as explained in Section 6.7.1), each instance of the Atomic Page Set Subsystem would *not* have its own, separate buffer areas. Instead, it would allocate buffer pages from a common Kernel Page Buffer Pool, and maintain "hints" lists concerning those pages which it most recently returned to the pool. For our purposes, however, it is more convenient (and not entirely inaccurate) to regard each instance of the Atomic Page Set Subsystem as having its own buffer areas.

The Permanent B-Tree Buffer holds the most recently accessed nodes (pages) of the Atomic Page Set Permanent B-Tree, for this logical disk. The Permanent B-Tree is the data structure used to record the sets of logical disk pages, comprising the current, permanent versions of the atomic page sets, stored on this disk. The logical structure of the Permanent B-Tree is identical to that of the Standard Page Set B-Tree, as discussed at length in Section 6.7.1, and illustrated in Figure 6-24. Refer to that Section for details. However, updating the Permanent B-Tree must be done more

---

[45] At this point, since *PSRestart* is assumed to have been invoked prior to *APSRestart*, all of the page allocation information for the logical disk has been reconstructed. Hence, it is now safe to allocate new logical disk pages, as required, without danger of reallocating pages that are already in use.

[46] Note that commit processing can be done "in the background" (after returning the result *val* to the invoker), just as in the case of *APSCommit*.

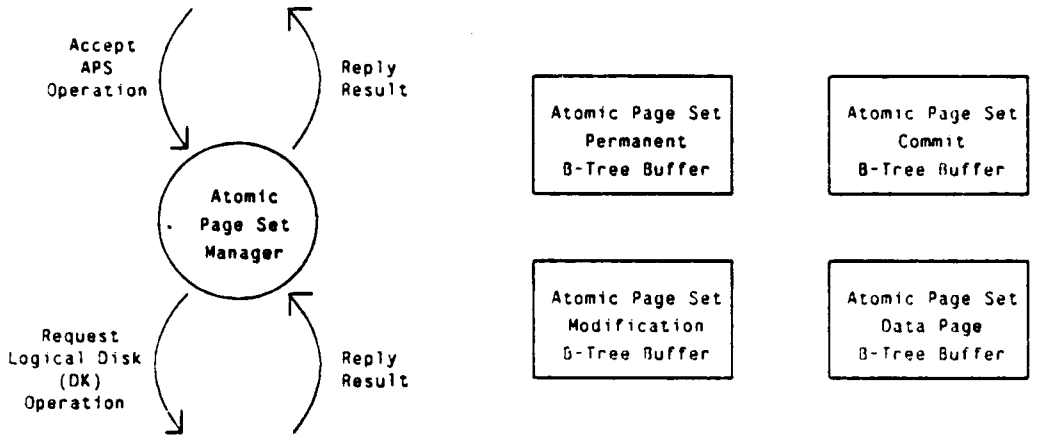


Figure 6-25: Components of the Atomic Page Set Subsystem

carefully than in the case of the Standard Page Set B-Tree, since in this case the update must be done atomically with respect to system failures. The technique for carefully (atomically) updating the Permanent B-Tree is outlined briefly in the discussion of transaction commit handling, below. As with the Standard Page Set B-Tree, a pointer to the *root node* of the Permanent B-tree (its *ldpnum*) is stored in page zero of the logical disk, so that the Atomic Page Set Subsystem can always find it.

The Modification B-Tree Buffer holds the most recently accessed pages of the Atomic Page Set Modification B-Tree. The Modification B-Tree is a temporary data structure, which contains the list of modifications that have been made to atomic page sets, in the course of the currently active, uncommitted transactions. Although the modification list is usually expected to be quite short, storing it as a B-tree will allow it to grow arbitrarily large, when needed. The general structure of the Modification B-Tree is similar to that of the Permanent B-Tree and the Standard Page Set B-Tree (see Figure 6-24). However, the nodes of the tree are stored in normal (rather than dual) pages, since this is a temporary data structure that will be "thrown away" automatically following a crash. The structure of a leaf node of the Modification B-Tree, along with its associated data pages, is illustrated in Figure 6-26.

The Modification List (B-Tree) is ordered by (*tid*, *psid*, *pnum*), and allows the mapping from these triples to the associated page modifications, to be performed very quickly. Page modifications are represented by a Data Page, containing the new values for the modified bytes within the page, and a Mask Page, which indicates which bytes have been modified. Modified bytes are indicated by a

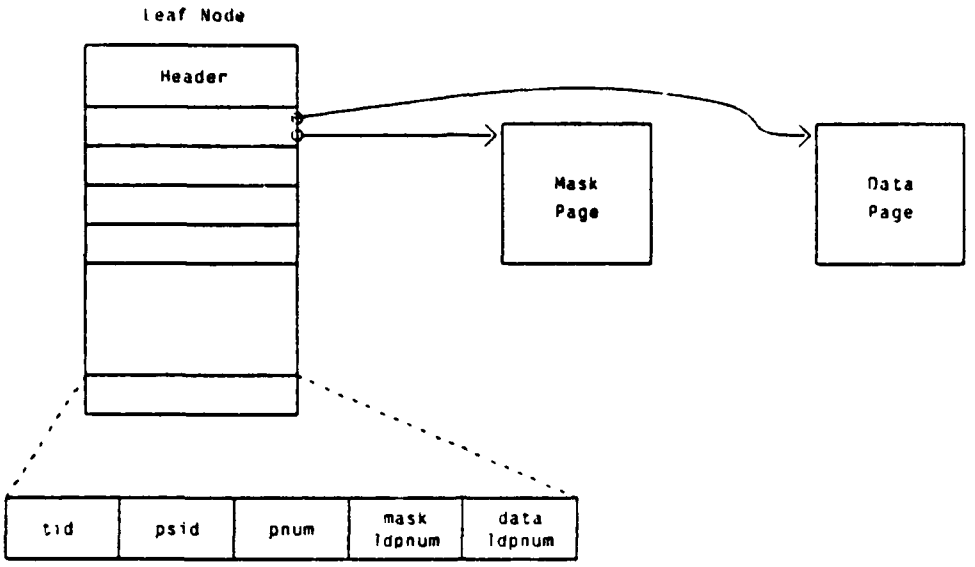


Figure 6-26: Atomic Page Set Modification List

corresponding Mask Page byte with hexadecimal value FF. Unmodified bytes have value zero in both the Data Page and the Mask Page.[47] If the entire Data Page has been modified, then the Mask Page can be omitted (*mask ldpnum = 0*). If the entire Data Page has been zeroed, then the Data Page can also be omitted (*data ldpnum = 0*). One other special case is that of atomic page sets that have been destroyed. In this case the Modification List contains only an entry for page zero of the destroyed page set, and *mask ldpnum = data ldpnum* = DESTROYED. Use of the Modification List when accessing and modifying the atomic page sets, and when committing or aborting transactions, is discussed below in Sections 6.7.2.2 and 6.7.2.3.

Another major data structure of the Atomic Page Set Subsystem, illustrated earlier in Figure 6-25, is the Commit B-Tree Buffer. It holds the most recently accessed pages of the Atomic Page Set Commit B-Tree, which is the data structure used to record the Commit List (the list of atomic page set modifications, made by transactions which are in the process of being committed). As with the Modification List, the Commit List is usually expected to be quite short. However, storing it as a B-tree will allow it to grow arbitrarily large, when needed. The structure of the Commit B-Tree is almost

[47] Although a bit map could be used in place of the current (byte map) Mask Page, the page oriented nature of the logical disk on which the maps are stored makes the current design more convenient. In addition, byte manipulations are usually more convenient and efficient than bit manipulations, in most current programming languages and hardware architectures.

identical to that of the Modification B-Tree, except that the nodes of the tree are stored in dual (rather than normal) pages. See Figure 6-26 above for an illustration of the structure of a leaf node of the Commit B-Tree. Updating the Commit B-Tree must be done carefully (i.e. atomically with respect to system failures), just as in the case of updating the Permanent B-Tree. This is to avoid any possible corruption or accidental loss of transactions, after they have been flagged as "prepared to commit". A pointer to the root node of the Commit B-Tree is stored in page zero of the logical disk, so that the Atomic Page Set Subsystem can always find it at restart time.

The final major component of the Atomic Page Set Subsystem is the Data Page Buffer. Unlike the Standard Page Set Subsystem, the Atomic Page Set Subsystem must often manipulate subparts (byte ranges) within data pages. The relevant pages are held in the Data Page Buffer while they are being operated on. Similarly, modification Mask Pages are constructed, accessed, and modified by the Atomic Page Set Subsystem, using the Data Page Buffer for temporary storage.

6.7.2.2 Accessing and Modifying Atomic Page Sets

Any access or modification to an Atomic Page Set must be done in the context of a transaction, specified by *tid*. All modifications made in the course of a particular transaction are first recorded (temporarily) in the Modification List. For example, assume that the following invocation has been made:

nbw = **APSWrite**(tid, psid, location, nbytes, buffer)

Then the Modification List will be searched for (*tid*, *psid*, *pnum*), where *pnum* is the page containing the bytes specified by *location* and *nbytes*.[48] If not found, a new entry with that key will be inserted in the list, with pointers to newly allocated (and completely zeroed) Mask and Data Pages. In either case, the specified bytes of the Data Page are set to the contents of *buffer*, and the corresponding bytes of the Mask Page are "turned on" (set to hexadecimal FF).

Accessing the current contents of an atomic page set is a little more involved. For example, assume that the following invocation has been made:

nbr = **APSRead**(tid, psid, location, nbytes, buffer)

Then *tid* must be taken into account when determining which modifications (if any) have been made to the specified bytes, and should be visible within this transaction. The Modification List is first searched for (*tid*, *psid*, *pnum*), where *pnum* is the page containing the bytes specified by *location* and *nbytes*. The Modification List is then successively searched for the ancestor transactions of *tid*, to see if any of them have modified the page in question. As relevant entries are found, a Compound

---

[48] To simplify our examples, we will omit the details concerning the handling of multiple pages, and the crossing of page boundaries.

Mask and Compound Data Page are constructed, such that the modifications made in subtransactions take precedence over their ancestors. This can be easily accomplished (at least conceptually) using bitwise logical operations on entire Mask and Data Pages. For example, to merge another (ancestor) Data Page (*Data*) and Mask Page (*Mask*) into the Compound Data and Mask Pages that have been constructed thus far, the following equations can be used:

CompoundData = CompoundData OR (Data AND (NOT CompoundMask))
CompoundMask = CompoundMask OR Mask

If, at any point in the construction of the Compound Data and Compound Mask Pages, it is found that the entire page has been modified, then there is no need to search any further. The requested bytes can be obtained directly from the Compound Data Page, and returned in *buffer*. Otherwise, the modifications indicted by the Compound Mask and Compound Data Pages (if any) must be applied (conceptually) to the corresponding page in the Atomic Page Set Permanent B-Tree. This is done using the same formula as shown above, where in this case "Data" is the Permanent Data Page. The requested bytes can then be obtain directly from the Compound Data Page (as before), and returned in *buffer*.

Note that, because of the buffering provided in the Atomic Page Set Subsystem, most manipulations of the Modification List, including Mask and Data Page manipulations, will not require actual disk operations. Furthermore, page-at-a-time access to atomic page sets will usually be much more efficient than manipulating a small number of bytes with each operation, and it can be made even more efficient by treating it as a special case.

6.7.2.3 Transaction Commit and Abort Handling

All modifications to atomic page sets are first made on a temporary basis, by recording them in the Modification List, as outlined above. A modification will only become "permanent" after its associated transaction (*tid*) has been committed. In the case of a nested *elementary* subtransaction, a modification can only become permanent after the top level ancestor transaction commits. This is because any failure (or abort) of an ancestor transaction, will cause the modifications made in any elementary subtransactions to also be aborted. When a nested elementary subtransaction (*tid*) commits, all of its modifications become visible to its parent transaction (*parent-tid*). This is accomplished by simply "renaming" all of the *tid* entries in the Modification List, to have transaction ID *parent-tid*. In case of conflicts, the modifications made in *tid* take precedence over the modifications made in *parent-tid*. This is done by constructing a Compound Mask and Compound Data Page, using the same equations as discussed above.

Committing a top level elementary transaction, or any compound transaction (whether nested or

not) causes all of its modifications to be permanently made to the affected atomic page sets. These modifications must be made very carefully, so as to preserve the atomic properties of the page sets (failure atomicity and durability). Since a transaction may span more than one logical disk, and more than one computing node, a two phase commit sequence is needed to coordinate the atomic updates of the various Atomic Page Set Subsystems, and to make them all occur as a single atomic update. In the first phase of the transaction commit sequence, the *APSPrepareCommit* primitive is invoked on each of the Atomic Page Set Subsystems involved in transaction *tid*. In response to *APSPrepareCommit*, each Atomic Page Set Subsystem must carefully record, in its Commit List, all of the modifications for transaction *tid*, so that they can no longer be lost in the event of a crash. This basically involves moving all of the entries with transaction ID *tid*, from the Modification List to the Commit List. However, to avoid inconsistencies in the Commit List, and the attendant loss of previously committed transactions, any updates to the Commit List must be done atomically with respect to system failures. The technique for atomically updating the Commit List B-tree is the same as that outlined below, for the case of the Atomic Page Set Permanent B-Tree.

After all of the Atomic Page Set Subsystems have indicated that they are prepared to commit transaction *tid*, the second phase of the commit sequence can begin. Phase two is signalled to each of the Atomic Page Set Subsystems by invoking the *APSCommit* primitive. In response to *APSCommit*, each subsystem simply flags transaction *tid* as committed, and replies that it has completed the request. This allows the invoker to continue its execution, while the Atomic Page Set Subsystem actually carries out the required commit processing. Flagging a transaction as committed is accomplished by writing its transaction ID (*tid*) into the header portion of the root node of the Commit List B-Tree. Note that since the nodes of the B-tree are stored as dual logical disk pages, updating the root node on disk will be an atomic operation.

Once a transaction has been flagged as committed, an Atomic Page Set Subsystem will not accept any other requests for operations on atomic page sets, until it has completed the necessary commit processing. Committing transaction *tid* basically requires carefully (atomically) updating the "permanent" contents of the atomic page sets (as stored in the Atomic Page Set Permanent B-Tree), based on the modification information contained in the Commit List. The possible types of modifications are: (1) the insertion of new pages (including entirely new page sets); (2) the modification of existing pages; (3) the removal (zeroing) of existing pages; and (4) the removal (destruction) of entire page sets. The key to making all of the modifications for transaction *tid* occur atomically, is to never actually modify any existing node or data pages in the Permanent B-Tree (except one). Instead new pages, reflecting the required modifications, are constructed as necessary. Then, all of the modifications can be made permanent at the same time, by atomically

updating a single Permanent B-tree node: the node at the root of the (minimum) subtree which encompasses all of the modifications. For more details concerning this technique, see [Mitchell 82], where the XDFS approach to atomically updating a file system B-tree is discussed.

After the Permanent B-Tree has been atomically updated, the modifications for transaction *tid* must be carefully removed from the Commit List, and *tid* erased from the header of the Commit List root node. This updating of the Commit List can all be done atomically, using the same technique as outlined above for the Permanent B-Tree. Note, however, that there is a period of time between the atomic updating of the Permanent B-Tree and the atomic updating of the Commit List, during which a system failure could occur. If a failure occurs during that time, the Atomic Page Set Subsystem will automatically (upon system restart) attempt to apply the modifications for transaction *tid* once again. This will not cause any problems, however, since all of the possible modifications to the Permanent B-Tree, as listed earlier, are *idempotent* (i.e. they can be repeated multiple times without changing the final result). When at last the Atomic Page Set Subsystem is able to complete the commit processing for transaction *tid*, it can then return to accepting new requests for operations on atomic page sets.

Besides committing, the other possible outcome for a transaction is that it aborts. A transaction (*tid*) can be aborted at any time, prior to being committed with *APSCommit*. The Atomic Page Set Subsystem is notified of transaction aborts by means of the *APSAbort* primitive. When a transaction is aborted, all of the modifications made within it, or any of its nested subtransactions (except already committed compound subtransactions), must be removed (undone). This is accomplished by removing all entries for transaction *tid*, and all of its subtransactions, from both the Modification List and the Commit List. Note that if entries are to be removed from the Commit List, it must be done atomically, using the standard technique outlined above.

6.7.3 Logical Disk Subsystem

Each physical disk that is being used by the ArchOS system has a corresponding Logical Disk Subsystem Kernel Arobject. These Arobjects are created and destroyed as disks are mounted and dismounted (see Restart/Reconfiguration Subsystem, Section 6.7.4). A physical disk can have at most one ArchOS partition on it, and only that portion of the disk will be used by ArchOS. We refer to the ArchOS partition of a disk as a *logical disk*. A logical disk can simply be viewed as a sequence of pages, where a page is (currently) defined to be 2K bytes in size. Pages on the disk are numbered from 0 to N-1, where N is the total number of pages on the logical disk. A logical disk has a unique identifier permanently associated with it, regardless of where the disk is currently mounted.

Each instance of the Logical Disk Subsystem is responsible for managing the allocation of pages on

its corresponding logical disk. It also keeps track of the bad (unusable) pages on the disk, so that they are not made available for allocation and later use. As much as possible, the Logical Disk Subsystem "optimizes" access to the physical disk, by allocating, reading, and writing multiple pages at a time, and ordering operations so as to minimize disk head movement. It also provides a degree of "network transparency", by automatically buffering and transferring data between the local and remote machines.

In addition to NORMAL (single) pages, the Logical Disk Subsystem also supports the concept of DUAL pages. A DUAL page is written more carefully than a NORMAL page, so that if a crash occurs while writing a DUAL page, its original contents can still be recovered. In this way, a simple, basic form of failure atomicity is provided. As the name suggests, each DUAL page is implemented using two single pages. DUAL pages are used for saving important data structures, and as a building block for implementing more complex atomic operations.

The Logical Disk Subsystem provides the following set of primitives:

```
pagelist = DKAllocate(npages, type [, follows])
val = DKFree(pagelist)

npr = DKRead(pagelist, buffer)
npw = DKWrite(pagelist, buffer)

val = DKSync()
val = DKStatus(statusbuffer)

val = DKRestart(dev-name)
val = DKRecoverDual(pagelist)

val = DKUnmark()
val = DKMark(pagelist)
val = DKGood()
val = DKBad(pagelist)
```

PAGELIST pagelist List of 32-bit logical disk page numbers, where the first 8 bits of a page number are used for various flags, and the remaining 24 bits are converted to the cylinder, track, and sector numbers of the physical disk. The first flag bit indicates whether the page is DUAL or NORMAL.

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

INT npr The actual number of pages read.

INT npw — The actual number of pages written.

INT npages — The number of pages to be allocated.

PAGETYPE type — The type of pages to be allocated: NORMAL or DUAL.

PAGEID follows — Logical disk page number, following which the new pages are to be allocated (physically as close as possible).

BUFFER *buffer — The buffer address in the kernel address space of either the local or remote node.

DKSTATUSBUFFER *statusbuffer
The buffer address for returning status information (can be either a local or remote kernel address).

DEVNAME dev-name
The logical device name.

The *DKAllocate* primitive allows a number of pages to be allocated at a time, and it returns a list of the allocated logical disk page numbers. An attempt is made to allocate pages which are physically close together on disk. The pages can be of type NORMAL or DUAL, and a preferred location on disk (near an existing page) can be specified. The *DKFree* primitive deallocates the specified list of pages.

DKRead reads a list of pages into the specified kernel buffer, and returns the count of pages actually read. Similarly, *DKWrite* writes a list of pages from the specified kernel buffer, and returns the count of pages actually written. When the buffer is in a remote kernel, *DKRead* and *DKWrite* automatically handle the cross node copying of data.

The *DKSync* primitive ensures that any buffered information within the Logical Disk Subsystem is consistent (synchronized) with the version on disk. The *DKStatus* primitive provides information about disk utilization, frequency of access, frequency of errors, number of bad blocks, and so on. It can be used, for example, by the Arobject/Process Management Subsystem, to determine the placement of process address space paging areas.

DKRestart is used only when the system is restarted. It initializes the disk hardware and any internal data structures. The *DKRecoverDual* primitive recovers the given list of DUAL pages, by ensuring that the two copies of each DUAL page are consistent (see [Sturgis 80] for details). This is used by various subsystems when recovering their data structures after a crash.

The *DKUnmark* and *DKMark* primitives are provided to help in garbage collection. First, *DKUnmark*

is used to flag all of the pages on the disk (except bad pages) as free. Then, *DKMark* is used (repeatedly) to flag the specified pages as allocated. Similarly, *DKGood* and *DKBad* are used for managing the bad pages (unusable pages) on the disk. *DKGood* flags all of the pages on the disk as good. Then, *DKBad* is used to flag pages that have been determined to be bad.

6.7.3.1 Components of the Logical Disk Subsystem

The Logical Disk Subsystem consists of a single Manager process, and three main data structures, which are shown in Figure 6-27. The Logical Disk Manager (LDM) process accepts all operations for the subsystem, executes them, and returns the results. The three data structures it uses are: (1)Sorted Page List, (2)Data Buffer, and (3)Page Allocation Map buffer.

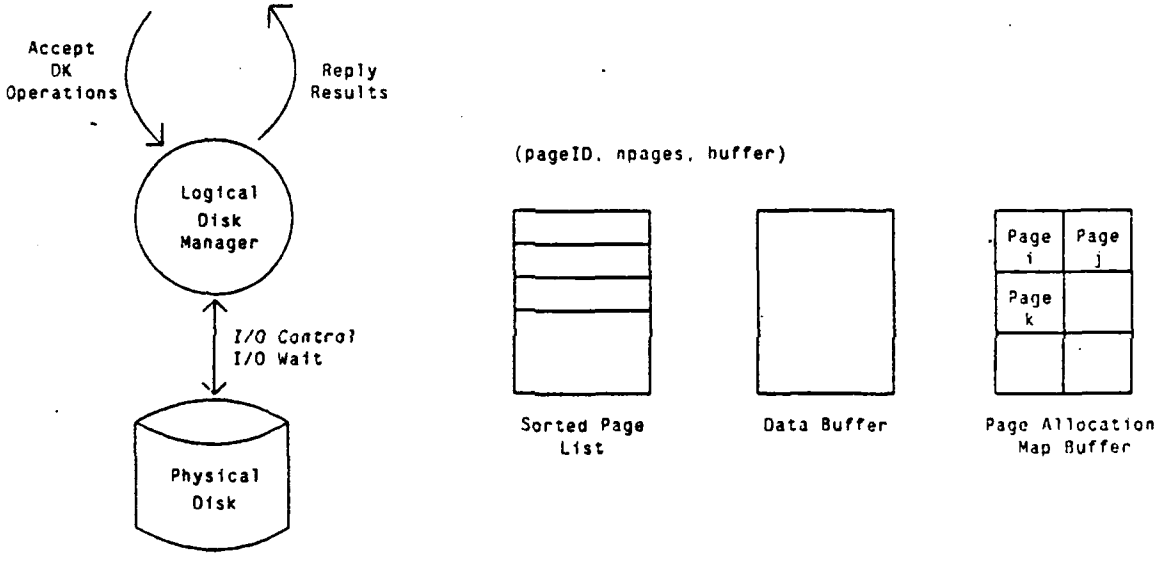


Figure 6-27: Components of the Logical Disk Subsystem

The purpose of the *Sorted Page List* is to order the disk pages to be read or written, so as to reduce disk head movement, and improve disk performance. Each entry of the Sorted Page List consists of three values: the logical page ID of the starting page, the number of pages (*npages*) to be read or written in one disk operation, and the location within the kernel buffer to be used for the operation. The Logical Disk Subsystem allows multiple page read and write operations. The *pagelist* provided is clustered (if possible) and then entered in the Sorted Page List.

The Data Buffer provided is used only for remote operations (invoked from other nodes in the

distributed system). For example, data read from the disk (for a remote operation) is first placed in the Data Buffer, and then transferred to the destination buffer on the remote node, by using the system *copy* facility. A "double buffering" technique is used, so that the disk transfer and *copy* operations can proceed in parallel.

The Page Allocation Map Buffer is used to buffer some of the pages of the Page Allocation Map. Whenever the Page Allocation Map has to be read or modified (such as in *DKAllocate*, *DKFree*, *DKMark*, and *DKUnmark*), the relevant pages of the map are first read into the buffer (unless they are already buffered). A detailed description of the Page Allocation Map and its handling is provided in Section 6.7.3.3.

6.7.3.2 Disk Layout

A logical disk is laid out as a sequence of pages, which are numbered from 0 to N-1, where N is the total number of pages on the logical disk. The *logical page number* specified in the logical disk operations consists of the concatenation of a *flags* byte, with a number from 0 to N-1. A page can either be NORMAL or DUAL. A NORMAL page is a single disk page, referred by a logical page number. A DUAL page consists of two pages, where the second page is located a fixed offset from the first (primary) page. The DUAL page is referred to by the logical page number of the primary page. The first bit in the *flags* byte (of the logical page number) indicates whether a page is NORMAL or DUAL. The two physical pages of a DUAL page have a gap of one page (pages are not adjacent), to reduce the probability of both pages being damaged by a single disk fault, and to ensure reasonably efficient sequential access.

Logical page 0 of the disk serves a special function. It contains information about the logical disk, and pointers to all important data structures saved on the disk. Thus, it saves information about the disk drive characteristics, the logical disk name, the logical disk size, and pointers to the Page Allocation Map, the Bad Page Table, the File System Directory, and so on. For reliability, page 0 is a DUAL page.

6.7.3.3 Disk Page Allocation

The Page Allocation Map is a bit map, which has one bit corresponding to each page on the logical disk. If a disk page has been allocated, its bit in the allocation map is set. The Page Allocation Map itself is saved near the middle of the disk, to provide efficient access. All pages of the allocation map are DUAL, and laid out as shown in Figure 6-28.[49]

---

[49] If we have a 500 MB logical disk, it would contain 250K pages of size 2K. The size of the allocation map would be 250K bits, or 16 DUAL pages (32 logical disk pages) approximately.

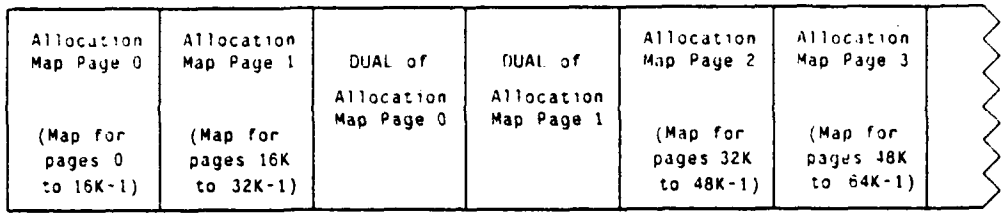


Figure 6-28: Layout of the Page Allocation Map

The page allocation algorithm used is fairly simple. It attempts to pack the allocated pages, starting at page 1 of the disk, and continuing through to page N-1. If the last page allocated was page *i*, the following allocation will start from page *i + 1* (unless a preferred allocation page is specified). The advantage of this scheme is that allocation and deallocation are very efficient. For allocation, the page of the allocation bitmap containing bit *i + 1* is likely to already be buffered, since that page was probably used recently (for the previous allocation operation). Hence, no disk reads will usually be required for allocation. For deallocation, the appropriate allocation map page can be easily determined, then read in (if necessary), and modified. Note that with this allocation scheme, successively allocated pages will tend to be located close together on disk, thus improving access efficiency.[50]

6.7.3.4 Bad Page Handling

One of the functions of logical disk management is to prevent the use of bad pages on the disk. A list, called the Bad Page List, is maintained near the end of the disk, and consists of the logical page numbers of all known, unusable (bad) pages.[51] The Bad Page List is stored as DUAL pages. When the Page Allocation Map is initialized (using *DKUnmark*), all the bad pages are marked as allocated, to prevent further usage of these pages.

The Bad Page List is usually constructed when a new disk is initialized, and is not expected to

---

[50] This allocation scheme will be able to cluster multiple page allocations reasonably well when the disk is new, but the disk will slowly become more and more fragmented with continued use. Some improvements could be made, to reduce the amount of fragmentation, if it is found to have a significant impact on performance.

[51] The entries in the Bad Page List are sorted when the list is first constructed, but new entries are simply added to the end of the list as found. The Bad Page List is expected to require only a few disk pages for storage. If 1% of the pages (size 2KB each) of a 500 MB disk are bad, the Bad Block List would have 2.5K entries. Given that each entry is 4 bytes long, 5 DUAL pages (10 disk pages) are required to save the entire list.

change very much after that. When the Bad Page List is constructed, the Data Buffer can be used for buffering it. The List will be constructed by a special utility, which first calls *DKGood* to remove the old Bad Page List (if it exists). After that, the entire disk is scanned to find any bad pages. This is done by writing each page and then reading it back. For each bad page found, its logical page ID is entered in the Bad Page List (with *DKBad*). Once the system is in operation, if a bad page is discovered by some subsystem, it can call *DKBad* to enter the page in the Bad Page List.

6.7.4 Restart/Reconfiguration Subsystem

The Page Set Restart/Reconfiguration Subsystem (PSR Subsystem) performs several functions related to the restart of a computation node following a crash, as well as the mounting and unmounting of logical disks. A single instance of this subsystem resides at each node in the system. The PSR subsystem "bootstraps" the entire Page Set Subsystem at restart time. After a crash, the Restart/Reconfiguration Subsystem is restarted automatically, and is responsible for constructing the entire Page Set Subsystem. The kernel (on a node) maintains a list of device addresses which it probes, to determine the logical disks on its own computing node. The PSR Subsystem sets up the atomic and standard page set arobjects, as well as the logical device arobject, for each logical disk found on the node.

In addition to restarting the Page Set Subsystem, the PSR Subsystem adds and removes logical disks dynamically (while the system is running) as necessary. When a new logical disk is added, the various Page Set Subsystem arobjects (Standard Page Set, Atomic Page Set, and Logical Disk) have to be created for it. When a disk is removed, the corresponding arobjects have to be killed. Some support is provided for the clean unmounting of disks. It is possible to quieten activity on a disk (by allowing old activity to complete, but refusing new activity), prior to disk removal.

The PSR Subsystem maintains a global Mount Table, which indicates the location (node) of all the logical disks in the entire distributed system, and the Page Set Subsystem Arobjects associated with each logical disk. Each instance of the PSR Subsystem maintains a copy of the Mount Table, and the peers co-operate in keeping up-to-date versions of the table. The Mount Table at a node is globally accessible to other kernel level subsystems on that node. Along with the global Mount Table, a local Unmount Table is also maintained. The Unmount Table lists the logical disks accessible to the PSR Subsystem, but not globally visible to other subsystems on its node, or to other nodes in the distributed system.

The PSR Subsystem performs two other useful functions. Firstly, it co-ordinates garbage collection for the entire Page Set Subsystem (on a node). Secondly, it checks any given disk for bad (unusable)

pages, and enters them in a Bad Page List (refer Section 6.7.3). Each of these two functions can be performed at restart, as well as when the system is running.

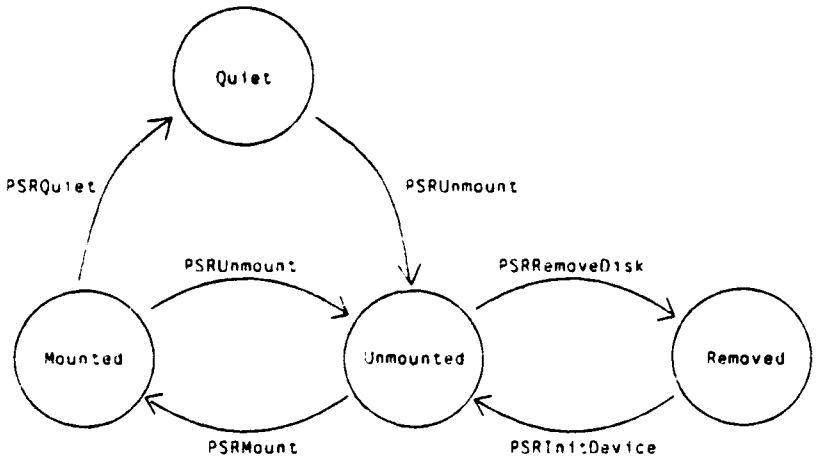


Figure 6-29: State Diagram for Logical Disks

The PSR Subsystem allows a logical disk to be in one of four possible states: Mounted, Unmounted, Quiet, or Removed (refer Figure 6-29). In the *Mounted* state, the logical disk has an entry in the Mount Table, and is globally visible on its own node, as well as throughout the system. All types of disk operations (except disk checking and garbage collection), such as paging, reading and writing, are permitted in this state. The *Quiet* state is very similar to the *Mounted* state in having an entry in the Mount Table, and being globally visible. The only difference is that this state represents an attempt to quiet down system activity prior to disk removal, and hence new activity is not allowed. In the *Unmounted* state, the logical disk has an entry only in the Unmount Table, and is visible within the PSR Subsystem. The main purpose of this state is to allow disk checking, garbage collection, and other disk maintenance to be performed, while the rest of the system is still up and running. The *Removed* state corresponds to the physical removal of the disk from the computer system.

val = **PSRRestart**(device-list [, options])
diskid = **PSRInitDevice**(dev-name [, initflags])
val = **PSRRemoveDisk**(diskid)
val = **PSRMount**(diskid)
val = **PSRUnmount**(diskid)
val = **PSRQuiet**(diskid)

val = **PSRCheckDisk**(diskid)
val = **PSRGarbageCollect**(diskid)

val = **PSRInsertMount**(diskid, *node*, disk-aid, ps-aid, aps-aid, *mtflags*)
val = **PSRRemoveMount**(diskid)
val = **PSRRequestMount**(mount-table)

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

DISKID diskid *The Logical Disk ID of the disk volume being operated on.*

DEVLIST device-list
The * of logical device names corresponding to logical disks in the system.

PSROPTIONS options
A set of boolean flags which can be specified as restart options for operations to be carried out in the restart sequence. The flags provided are: GARBAGE-COLLECT, and CHECKDISK.

DEVNAME dev-name
The logical device name.

PSRINITFLAGS initflags
A set of boolean flags which can be specified when a logical disk is initialized. Currently, two flags are provided: READ-ONLY, and SELF-CONTAINED.

NODENAME node The ID of the node on which the disk is mounted.

AID disk-aid The arobject ID for the Logical Disk Subsystem Arobject corresponding to the disk *diskid*.

AID ps-aid The arobject ID for the Standard Page Set Subsystem Arobject corresponding to the disk *diskid*.

AID aps-aid The arobject ID for the Atomic Page Set Subsystem Arobject corresponding to the disk *diskid*.

PSRMOUNTFLAGS mtflags
A set of boolean flags associated with each entry in the Mount Table. The flags provided are: QUIET, READ-ONLY, SELF-CONTAINED.

MOUNTBUFFER *mount-table
: The buffer used for transferring the contents of the mount table between different nodes in the system.

The *PSRRestart* primitive restarts the Page Set Subsystem on a given node, following a node crash. It is invoked by a kernel process responsible for bringing up the entire kernel on that node. The list of devices corresponding to all logical disks on the node is determined by the kernel process, and supplied as a *PSRRestart* parameter. For each logical disk on the node, the *PSRRestart* primitive creates and restarts three page set subsystem arobjects (standard page set, atomic page set, and logical disk). In addition, if the GARBAGE-COLLECT and/or CHECKDISK options are specified, then garbage collection and/or disk checking (for bad pages) are performed as part of the page set restart sequence.

The *PSRInitDevice* primitive adds a new logical disk to the node. The state of the logical disk makes a trasition from the *Removed* state to the *Unmounted* state. The system device name (*dev-name*) for the logical disk is supplied, and its logical ID is found (from page 0 of the disk) and returned. The Standard Page Set, Atomic Page Set, and Logical Disk arobjects are created for the newly added disk. Optional flags (READ-ONLY, and SELF-CONTAINED) can be specified as necessary, for restricting the type of disk access permitted. The READ-ONLY flag specifies that the entire disk is read-only, and its contents cannot be modified. The SELF-CONTAINED flag specifies that all files saved on that disk must have the directory entries, as well as the file contents (page sets) saved on that disk.[52] The *PSRRemoveDisk* primitive removes a specified *unmounted* logical disk from the system. The three page set arobjects for the logical disk are destroyed.

The *PSRMount* primitive mounts the specified unmounted logical disk, and makes it globally accessible. All nodes in the distributed system are informed about the newly mounted disk. The *PSRUnmount* primitive unmounts the specified logical disk, from the *Mounted* or *Quiet* state. In either case, the result of a *PSRUnmount* is seen by the rest of the system as a disk crash, where no further operations are permitted, and ongoing operations cannot be completed.[53] It is expected, however, that if the disk was *quiet* prior to unmounting, there would be very little (if any) ongoing activity at this time. The *PSRQuiet* primitive flags a mounted logical disk as being in a special *quiet* state, in which

---

[52] If a disk is self contained, it can be easily moved from one ArchOS system, and initialized on another.

[53] *PSRUnmount* can be implemented as the destruction of the three page set arobjects (standard page set, atomic page set, and logical disk), followed by the creation of new page set arobjects. In this case, all requests for operations with the old page set arobject IDs will return error indications.

ongoing activity can be continued, but new activity is not allowed. The main prupose of this primitive is to allow for clean disk unmounting.

The *PSRCheckDisk* and *PSRGarbageCollect* primitives are allowed only when the specified logical disk is in the *Unmounted* state. The *PSRCheckDisk* primitive determines the bad pages on the specified disk (non-destructively), and constructs the Bad Page List. The *PSRGarbageCollect* primitive co-ordinates garbage collection for the specified disk, by invoking garbage collection primitives provided by the standard page set, atomic page set, and logical disk arobjects.

The Mount Table is replicated on all nodes in the distributed system. Three primitives are provided, which are used only by peer Restart/Reconfiguration arobjects on other nodes, for obtaining and updating information from the Mount Table. The *PSRInsertMount* primitive is used for adding a new Mount Table entry, or for modifying an existing entry. The parameters specified are the logical disk ID (*diskid*), the ID of the node on which the disk is mounted (*node*), the IDs of the three Page Set Subsystem arobjects: logical disk subsystem (*disk-aid*), standard page set (*ps-aid*), and atomic page set (*aps-aid*), and a set of flags. The *PSRRemoveMount* primitive removes the Mount Table entry for the logical disk. Finally, the *PSRRequestMount* primitive returns the contents of the entire Mount Table in the buffer provided. This primitive is primarily invoked by a peer PSR arobject which is restarting after a crash, and attempting to reconstuct its copy of the system-wide Mount Table.

6.7.4.1 Components of the Restart/Reconfiguration Subsystem

The PSR Subsystem consists of four types of components: (1)the Mount Table, (2)the Unmount Table, (3)the PSR Manager process, and (4)one or more PSR Worker processes. Each of these components is briefly described below, and shown in Figure 6-30.

The Mount Table provides information about the location of all (*mounted* and *quiet*) logical disks in the entire distributed system, and the arobjects which are responsible for managing these disks. A copy of the Mount Table is maintained at each node in the system. The Mount Table is globally visible to all subsystems on a node (and to other nodes), but is updated only by the PSR Subsystem. During updates, the table is locked, to prevent the access of inconsistent states of the table. Each entry of the Mount Table contains the following items of information: the ID of the logical (*mounted* or *quiet* disk, the ID of the node on which it is mounted, the ID of the Logical Disk Subsystem arobject for that disk, the ID of the Standard Page Set arobject for that disk, the ID of the Atomic Page Set Arobject for that disk and a set of flags (QUIET, READ-ONLY, and SELF-CONTAINED). The QUIET flag is the only way of specifying whether a logical disk is in the *mounted* state, or in the *quiet* state. The READ-ONLY and SELF-CONTAINED flags specify particular types of disk usage, and has been discussed in the previous section.

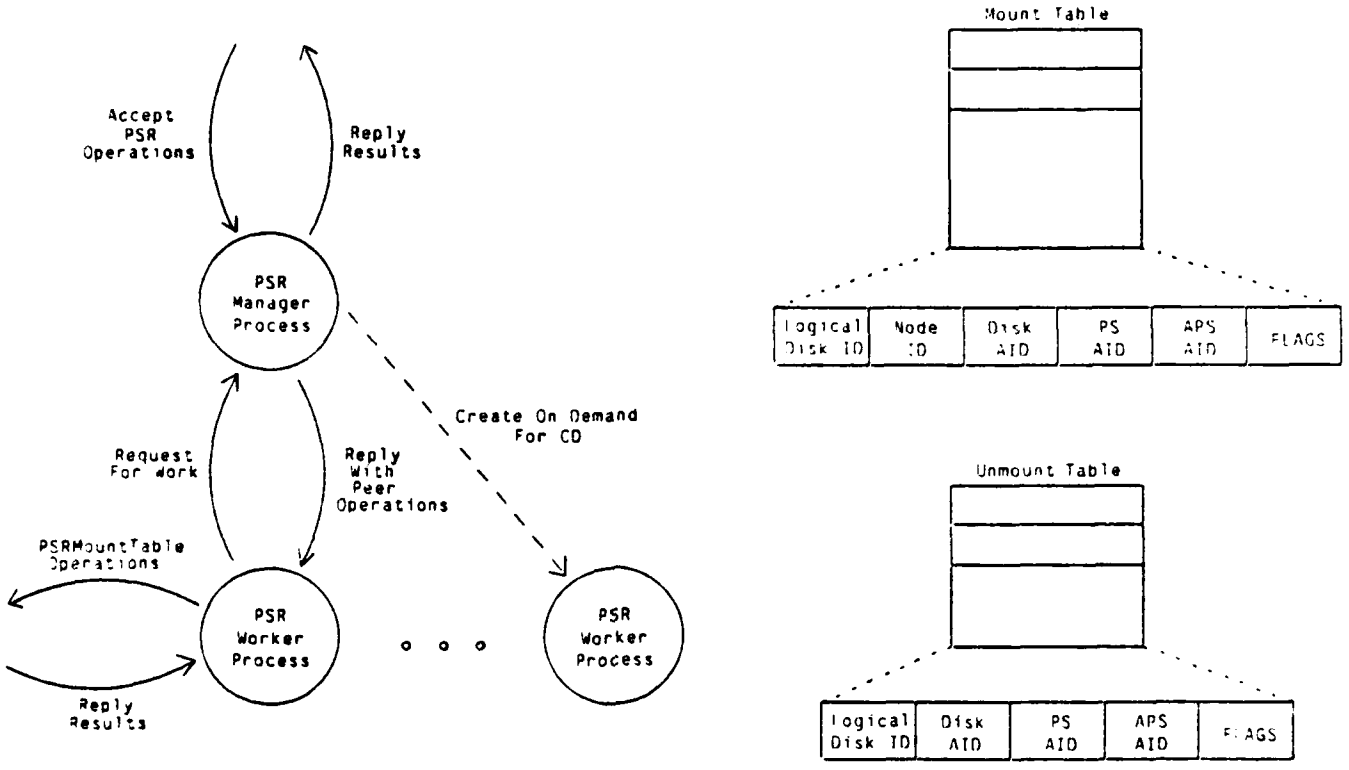


Figure 6-30: Components of the Restart/Reconfiguration Subsystem

The Unmount Table provides information about *unmounted* logical disks, which can be accessed by the PSR Subsystem, but are not visible to the rest of the system. It is useful to be able to access disks in this mode, primarily for maintenance purposes, such as disk checking and garbage collection. The entries in the Unmount Table are very similar to those in the Mount Table, except that the ID of the node is not required, since the Unmount Table refers only to its local node. In the flags field, the QUIET flag is not required for this table.

The PSR Manager process is responsible for providing most of the functionality of the PSR Subsystem. It accepts all PSR operations, and returns the results. It also manages the Mount and the Unmount tables. The PSR Worker processes serve two functions. They implement *PSRCheckDisk* and *PSRGarbageCollect* operations on behalf of the Manager. Since both these operations can potentially take a very long time, the PSR Subsystem can still be used for performing operations on other disks concurrently. The other function of the Worker process is to wait on behalf of the

Manager process, when peer level operations are invoked by the Manager. This prevents deadlocks arising from multiple, concurrent peer operations.[54] One Worker process is always present in the PSR Subsystem for invoking peer PSR operations (PSRInsertMount, PSRRemoveMount, and PSRRequestMount). For each *PSRcheckDisk* or *PSRGarbageCollect* operation, a new Worker process is created, and then destroyed when the operation is completed.

6.7.4.2 Restart

The PSR Subsystem performs two related functions when restarting after a node crash. The Mount Table has to be recreated, and the Page Set Subsystem has to be brought up. The reconstruction of the Mount Table is quite similar to the reconstruction of the Directory Map Table, when the Directory Map Subsystem of the File System is restarted (refer Section 6.6.3). The PSR Manager process invokes a Worker process to obtain the Mount Table (with *PSRRequestMount*) from one of the other nodes in the system. In the meantime, it proceeds to initialize all the logical disks on its own node, which have been found by the kernel restart process. It determines the IDs of the logical disks, and mounts them.

For each logical disk on the node, its ID is first determined, and an entry made in the Unmount Table. Next, a Logical Disk Subsystem arobject is created for that disk, and then restarted. As a result of the Disk Subsystem restart, all the logical disk data structures are recovered (page 0 of the disk, the allocation map, and the bad page list). Once this operation has completed, the standard page set arobject is created and restarted, so that the Page Set B-tree is recovered correctly. Next, the atomic page set arobject is created and restarted. This allows all its data structures to be recovered, and all committed transactions to be completed. The IDs for the three newly created arobjects are entered in the Unmount Table. Next, the logical disk is *mounted*, and entered in the Mount Table. Once all the logical disks are thus mounted, all the peer PSR arobjects are told to update their Mount Tables, using the *PSRInsertMount* primitive.

If the disk checking and garbage collection options are specified, then these operations also have to be performed as part of the restart sequence. Garbage collection is a good idea after a restart, so that all temporary page sets can be flushed.

---

[54] Note the similarity with the handling of the Directory Map Table in the File Subsystem in Section 6.6.3.

6.7.4.3 Garbage Collection and Disk Checking

The PSR Subsystem co-ordinates garbage collection for the Page Set Subsystem. It first calls on the *DKUnmark* primitive, so that the Logical Disk Subsystem marks the entire allocation map as being free. Next, it calls the *PSGarbageCollect* primitive, which specifies all the pages being used by the Standard Page Set Subsystem. Following that, the *APSGarbageCollect* primitive is invoked, which marks all the disk pages in use by the Atomic Page Set Subsystem. As a result of this garbage collection procedure, all temporary page sets, as well as pages not in use by the Page Set Subsystem are freed.[55]

Disk checking determines all the unusable (bad) pages on the specified disk in a non-destructive manner. First, the *DKGood* primitive is called to remove the old Bad Page List, thereby marking all disk pages as "good". Then each page on the disk is read. If there is no checksum error in reading, then the page is usable. If there is an error, the page is written into, and then read again. If the error persists, then the page number is entered in the Bad Page List. All the pages of the disk are checked in this way.

6.8 I/O Device Subsystem

An I/O device is treated as a special type of file where no transaction is allowed and a device specific control scheme is included. To read, write or issue a special command such as reset, a device must be opened before any read/write access and it has to be closed after all of the actions are completed.

All of the device dependent commands can be sent to devices by using the SetIoctl primitive.

```
dd = OpenDevice(devicename, mode)
CloseDevice(dd)

nr = ReadDevice(dd, buf, nbytes)
nw = WriteDevice(dd, buf, nbytes)

event-cnt = Iowait(dev-descriptor, timeout)

val = SetIoctl(dev-descriptor, io-command, dev-buf, timeout)
```

---

[55] It is still possible for unreferenced files to exist, since this garbage collection procedure does not consider subsystems above the Page Set Subsystem.

It should be noted that the current I/O system does not interact with the transaction manager at all. Thus, any type of transaction facility is provided.

6.8.0.1 Policy Management

The policy management provides system functions to add, delete, and modify the policy definition module in ArchOS. Since the placement of the *policy definition module* is a major issue in terms of the system performance, ArchOS allows a client to specify the location by using a *policy definition descriptor*. The policy definition module consists of a *policy body* and a set of *policy attributes*. Both the policy body and attributes can be modified at runtime.

---

pdd = **AllocatePDD**()
val = **FreePDD**(pdd)

val = **SetPolicy**(policy-name, policy-def-desc)
val = **SetAttribute**(policy-name, attribute-name, attribute-value)

An *AllocatePDD* primitive allocates a policy definition descriptor in the kernel and *FreePDD* releases the allocated descriptor. A *SetPolicy* primitive links a user-defined policy definition module to ArchOS. A *SetAttribute* primitive set a specific value(s) for its one of attribute.

6.9 Time-Driven Scheduler Subsystem

Process scheduling in a real-time facility is crucial to the success of the system. Process scheduling in ArchOS differs fundamentally from scheduling on other operating systems because of several critical factors:

- ArchOS is a real-time system; hence process scheduling must be compatible with its critical goal of supporting application-defined time constraints
- ArchOS design manages application time-constraints by explicitly accounting for the fact that there is a definable time-varying value to the system for completing each process at a particular time.

6.9.1 Best-Effort Scheduling

6.9.1.1 Value Function Processing

The ArchOS scheduler, part of the ArchOS Kernel Arobject (see Section 6.3.1 designed to explicitly use the application defined value function for the set of processes in its queue in making a "best effort" decision on the process to be executed at each point in time.

The computational model for this scheduler consists of a set of schedulable processes p_i resident in a processor. Each such process has a request time R_i, an estimated computation interval C_i and a value function $V_i(t)$, where t is a time for which the value is to be determined. Figure 6-31 illustrates these process attributes for a process with a linearly decreasing value function prior to a critical time D_i, and an exponential value decay following the critical time. The illustration depicts a process which is dispatched after its request time and which completes prior to its critical time without being preempted.

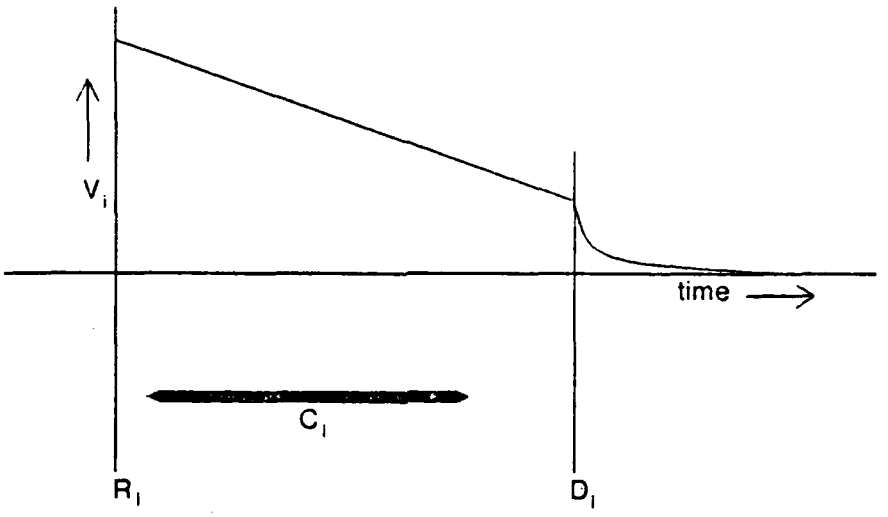


Figure 6-31: Process Model Attributes for Process i

$V_i(t)$ defines the value to the system due to completing p_i at time t. The set of these value functions is used by the scheduler to determine the best sequence in which to schedule each of the available processes. The type of functions definable for V_i will determine the range of scheduling policies supportable by the operating system, particularly with respect to the handling of a processor overload in which some critical times cannot be met.

We note that the existence and importance of the deadline is dependent on the value function. The value function can be said to define an explicit deadline only if it has a discontinuity in the function, its first derivative, or its second derivative, in which the value is lower or decreasing after the discontinuity. In Figure 6-31, the deadline is defined by the discontinuity in its first derivative at the critical time D_i.

At any particular point in time, there will be n processes ready for scheduling, resulting in n! possible scheduling sequences. Each of these sequences consists of a process ordering

$(m_1, \cdots, m_n)$, where p_{m_j} would be the j^{th} process to be scheduled. A scheduling sequence will be considered optimal if, with respect to the available information at the time of the scheduling decision, β is maximized, where $\beta = \sum V_i(T_i)$ and T_i is the actual completion time of p_i using this scheduling sequence (i.e., if p_i is the j^{th} process to be scheduled, then $T_i = \sum_{k=1}^{j} C_{m_k}$). See Figure 6-32 for an example of four processes with value functions, for which the choice of a best effort schedule is non-trivial. The figure shows the four value functions, and a potential scheduling sequence such that each completes with a high value.

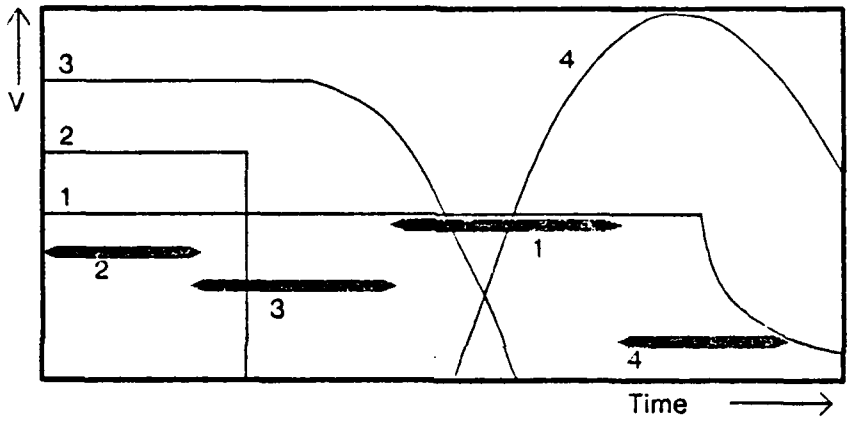


Figure 6-32: Four "Typical" Processes with their Value Functions

Since the completion times used for this scheduler will be known only stochastically, the assumed distribution and its computed parameters (e.g., mean, variance) will be used to compute its expected value in the scheduling sequence, resulting in a statistically "good" sequence. Making an optimum schedule can be shown to be computationally intractable, so the best-effort scheduler will use a set of heuristics to determine a sequence which will generate a high total value over any time period which is sufficiently long with respect to the completion times of the processes to be scheduled.

6.9.1.2 Well-Known Scheduling Algorithms

Scheduling a set of processes consists of producing a sequence of processes on one or more processors such that the utilization of resources optimizes some scheduling criterion. Criteria which have historically been used to generate process schedules includes maximizing process flow (i.e., minimizing the elapsed time for the entire sequence), or minimizing the maximum lateness (lateness is defined to be the difference between the time a process is completed and its deadline). It has long

been known [Conway 67] that there are simple algorithms which will optimize certain such criteria under certain conditions, but algorithms optimizing most of the interesting scheduling criteria are known to be NP-complete [Garey 79], indicating that there is no known efficient algorithm which can produce an optimum sequence. Clearly, the choice of metric is crucial to the generation of a processing sequence which will meet the goals of the system for which the schedule is being prepared.

There are several well-known scheduling algorithms which have traditionally been used in process scheduling, each with properties which make it useful for certain applications. Among these are:

1. **SPT**. At each decision point, the process with the shortest estimated completion time is executed. This algorithm maximizes overall throughput, and is frequently used (although in modified form) in batch systems.

2. **Deadline**. At each decision point, the process with the earliest deadline is executed. This algorithm, in the absence of an overload, will result in meeting all deadlines. More precisely, this algorithm will minimize the maximum lateness.

3. **Slack**. At each decision point, the process with the smallest estimated slack time (elapsed time to the critical time minus its estimated completion time) is executed. This algorithm, in the absence of an overload, will also result in meeting all deadlines, but will produce a much higher level of preemptions. This algorithm will maximize the minimum lateness.

4. **FIFO**. At each decision point, the process which has been in the request set longest is executed. This algorithm will produce a relatively "fair" schedule, in which lateness will be spread out to all processes.

5. **Priority**. At each decision point, the process with the highest fixed priority is executed. The "most important" process is executing at any moment.

Of these, real-time systems traditionally use only the Priority and/or FIFO schedulers. It should be noted that no objective performance measures (e.g., meeting deadlines, high throughput, low lateness or tardiness) are even approximately optimized by these algorithms, but they are inexpensive to implement and require very few run-time resources.

In priority-driven scheduler systems, deadline management is attempted by assigning a high fixed priority to processes with "important" deadlines, disregarding the resulting impact to less "important" deadlines. During the testing period, these priorities are (usually manually) adjusted until the system implementer is convinced that the system "works". This approach can work only for relatively simple systems, since the fixed priorities do not reflect any time-varying value of the computations with respect to the problem being solved, nor do they reflect fact that there are many

schedulable sets of process deadlines which cannot be met with fixed priorities. In addition, implementers of such systems find that it is extremely difficult to determine reasonable priorities, since, typically, each individual subsystem implementer feels that his or her program is of high importance to the system. This problem is usually "solved" by deferring final priority determination to the system test phase of implementation, so the resulting performance problems remain hidden until it is too late to consider the most effective design solutions.

Using a deadline scheduler(i.e., a scheduler which schedules the process with the closest deadline first [Liu 73]) solves the problem of missing otherwise schedulable deadlines due to the imposition of fixed priorities, but leaves other problems, most notably the problem of the transient overload. The deadline scheduler provides no reasonable control of the choice of which deadlines must be delayed in an overload, leading to unpredictable failures and resulting in an impact on reliability and maintainability.

6.9.1.3 A Best-Effort Scheduling Algorithm

The creation of a best-effort scheduling algorithm is one of the key research interests of the Archons project, and work on it is currently underway at this writing. However, there are some preliminary results which have been produced, and which will be used, at least in our initial scheduler design. In this initial algorithm, we take advantage of several observed value function and scheduling characteristics:

- Given a set of processes with deadlines *which can all be met* based on the sequence of the deadlines and the computation times of the processes, it can be shown that a schedule in which the process with the earliest deadline is scheduled first (i.e., a *Deadline* schedule) will always result in meeting all deadlines.

- Given a set of processes (ignoring deadlines) with known values for completing them, it can be shown that a schedule in which the process with the highest value density (*V/C*, in which V) is its value and C) is its processing time) is processed first will produce a total value at any period of time no lower than any other schedule.

- Most value functions of interest (at least among those investigated at this time) have their highest value occurring immediately prior to the critical time.

Some of the implications of these observations are:

- If no overload occurs, all deadlines will be met, and the value function produced by the deadline schedule will be as high as possible.

- If an overload occurs, and some processes must miss their deadlines, the Value Density Schedule would produce a high value.

- Therefore, if we can predict the probability of an overload, we can choose processes with

low value density as candidates for being removed from a deadline schedule, until a deadline schedule is produced which has an acceptably low probability of producing an overload.

The algorithm we will use, then, will start with a deadline-ordered sequence of the available processes, which will be sequentially checked for its probability of overload. At any point in the sequence in which the overload probability passes a preset point, the process prior to the overload with the lowest value density will be removed from the sequence, repeating until the overload probability is acceptable.

6.9.2 Time Management

The ArchOS scheduler will use information provided by the time management primitives to make its scheduling decisions. This information is derived from these primitives:

```
rtc = GetRealTime()
Timedate(time, date)

TimeDate = Delay(deltime, criticaltime, comptime, setflag)
TimeDate = Alarm(alarmtime, criticaltime, comptime, setflag)

val = SetTimeDate(time, date)
```

In addition to these primitives, policy directives will be used which are provided by the Policy module (see Section 4.2.6).

The information needed for making the scheduling decisions includes:

1. **The request time.** This is the time a process becomes available for execution, although its value function may be negative, forcing the scheduler to delay it until it can be completed with as high a positive value as possible. Its value function is defined by the process itself using the **delay** or **alarm** primitives.

2. **The critical time.** This time is the time relative to the request time for the process at which the value function may have a discontinuity, and is determined when the process was requested. Its value relative to the request time is set by the process itself using the **delay** or **alarm** primitives.

3. **The value function parameters themselves.** The value function is divided into two parts: (1) the value for completing the process prior to its critical time, and (2) the value for completing the process after its critical time. In each of these functions, the time used in computing the function is relative to the critical time, so the value function used prior to the critical time measures its time "backwards". Each function has the form: $V(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K5t}$) so there are ten parameters defining the total value function.

Value function parameters are defined by the application programmer, but are modified by the system policy currently in effect. In the case of processes (i.e., server processes) scheduled on behalf of other processes (i.e., client processes), the value function can, at the option of the process, use either the value function of its client or its own value function.

4. **System policy.** The system policy is set by the policy module, and consists of a set of specific policy choices (e.g., abort processes whose value functions have fallen below zero) as well as two parameters for each process in the system. The values are K_a and K_b, and are used to make a linear transformation to the application-defined value function: $V'(t) = K_a + K_b V(t)$) It is $V'(t)$ which is actually used to perform the value function scheduling computation.

6.9.3 Short-term vs. Long-term Scheduling

Using the algorithm described above in subsection 6.9.1.3, these decisions will result in a long-term high value as long as an overload condition is not maintained for a significant period of time. A critical part of the scheduling algorithm will be the computation of the probability that an overload condition, other than a transient overload, has occurred or is about to occur, which will result in making a decision about long-term reconfiguration across the system node boundaries. In addition to decisions about reconfiguration, decisions with respect to process abortion, preemption, and process scheduling in support of client processes outside the node will be made by the scheduling algorithm.

ArchOS will use a three phased approach to handling the inter-node scheduling control decisions, including reconfiguration decisions as well as decisions about scheduling processes invoked across node boundaries in support of common tasks, and including such decisions as when a process should be moved and the decision of the best destination to which it should be moved.

1. Purely local information will be utilized. This means that the attributes of the functions resident on the node will be used, but no inter-node coordination of process scheduling will take place. Decisions by processes in one arobject to request services in another arobject (either local or remote) will be made unilaterally by the requesting node.

2. Primarily local information will be utilized, augmented with value function and critical time information from the clients of the processes being scheduled. In this way, scheduling by one node on behalf of another node will be supportive of the needs of the client node as far as possible, but no coordination of, for example, two server nodes in support of a third client node will be performed. Thus, it is possible (but hopefully unlikely) that in a heavily loaded environment two such server nodes could abort processes on behalf of clients in such a way that no client gets adequately served, even though work is being performed on its behalf.

3. Local information will be augmented by some form of negotiation among scheduling routines to ensure that work on behalf of remote client processes is coordinated, and, if abortion is necessary, a best-effort is made to do it in a way which maximizes the total system value.

6.10 Time-Driven Virtual Memory Subsystem

The Time-Driven Virtual Memory (TDVM) Subsystem is responsible for managing the primary memory page frames. It allocates those page frames among the most critical processes, as dynamically determined by the Time-Driven Scheduler (TDS) Subsystem. The basic goal of the TDS and TDVM Subsystems is to ensure that the required CPU time and primary memory space resources are both available when needed, in order to complete critical tasks within their deadlines. Furthermore, when it becomes necessary to miss some deadlines due to overload situations, TDS and TDVM attempt to allocate the oversubscribed resources to the most important processes, so as to maximize the total value of the tasks completed (see Section TDSSEC).

The basic goal of the TDVM Subsystem differs substantially from that of a general purpose time-sharing system's virtual memory manager. Although there are many theoretical results and a great deal of practical experience in designing virtual memory systems for general purpose computing environments, it is unclear how much of this work can be applied in the real-time (or time-driven) environment of ArchOS. Indeed, very few real-time systems to date have supported virtual memory facilities, since it is very difficult to provide real-time guarantees in the face of unexpected paging activity. Thus, the design of the ArchOS TDVM Subsystem, as outlined in this Section, can only be regarded as a preliminary version. It provides the required functionality, but many of the policies and design decisions are based on little or no supporting data or theory. This area of the ArchOS design will be one of the important focus points for further study, experimentation, and development.

6.10.1 Memory Management Policies and Techniques

In order to manage the primary memory (space) resources in a time-driven fashion, consistent with the management of the CPU (time) resources, the TDVM Subsystem uses the "working set model" as the basis for its management policies and techniques. The working set model for memory management was first described by Denning [Denning 68], and it has been the focus of a great deal of subsequent research [Denning 80]. The working set of a process is defined as the set of its virtual memory pages which have been accessed within the last W units of CPU (virtual) time. Note that working sets are defined on a per-process basis, and are determined with respect to the per-process virtual times. The goal of a working set memory manager is to ensure that the complete working set for each active process is resident in primary memory. If there are too many active processes, some will have to be swapped out to secondary memory (and thus become inactive), to ensure that all of the remaining working sets will fit within primary memory. This guarantees that the active processes will execute efficiently, with a minimum number of page faults (i.e. "thrashing" is avoided).

Since a working set memory manager determines which pages are to be resident in primary memory on a per-process basis, it is called a "local" memory management technique. This is in contrast to "global" techniques, which are only concerned with the *real* time since a page was last accessed, regardless of which process it belongs to. For the purposes of time-driven virtual memory management, a local technique is expected to be much more effective in allocating the primary memory resources to the most important processes. This is because it can account for variations in the urgency of processes, based on their deadlines and the penalties for missing them, and ensure that the most urgent processes have their working sets completely resident.

A key factor affecting the operation of a working set memory manager is the choice of W, the working set parameter or working set window size. In traditional time-sharing environments, it has been found that using a single value of W, the same for all processes, works almost as effectively as selecting the window size on a per-process basis. Also, the overall paging behavior of the system is not very sensitive to small changes in W, although W must be tuned somewhat to yield performance that is close to optimum.[56] The situation in a time-driven environment is somewhat different, since the goal is to ensure that deadlines, especially all of the more important deadlines, are met, and there is much less concern with overall system throughput. In this case it could be beneficial to vary the working set window sizes, depending on the criticality of the individual processes. Furthermore, since the criticality of a process can vary with time, it may be useful to dynamically vary its window size as well.

Since the value of W depends upon the criticality of the individual process (how near it is to its deadline, and the penalty for missing it), only the TDS Subsystem is in a position to specify the varying window sizes. Only a few, discrete window sizes, corresponding to multiples of some standard window size, w, should actually be needed. An adequate set of sizes might be $\{w, 2w, 4w, \ldots, 128w\}$. Note that w is a global system tuning parameter, corresponding to the single window size of a standard, time-sharing, working set memory manager. TDS informs TDVM of the various working set window sizes at the same time it provides TDVM with the ordered list of runnable processes, i.e. as a result of calling the *Schedule* primitive (see Section TDSKERSEC).

For the case of time-sharing working set memory managers, it has been found that maintaining just the working set sizes (number of pages) across process swaps, without recording the individual

---

[56] If W is too small excessive paging will occur, because it will not capture the "natural" working sets for many processes. On the other hand, if W is too large, excessive swapping will occur. This is because the working sets will be larger, due to pages taking longer to pass out of the working set window, and fewer working sets can then be fit in primary memory. Getting W close to optimum involves balancing these two effects so as to maximize overall system throughput.

pages of the working sets, still yields adequate performance. The working set pages are simply faulted back in on demand, after the process has been swapped back in. However, in a time-driven system such paging behavior may not be acceptable, especially if the process is nearing its deadline. As a result, TDVM will maintain the actual lists of working set pages for all processes, whether active or swapped. This will allow the working set of a process to be efficiently reloaded at swap-in time.

The choice of which working sets should be resident in primary memory, and which should be swapped-out, will be made by TDVM based on the order of the processes in the scheduling list, which is returned by the TDS *Schedule* (or *RequestSwapList*) primitive. In general, the most urgent processes will be at the head of the scheduling list, and TDVM will attempt to load and maintain the working sets of as many of those processes as possible. Since it is expected that most processes will gradually rise through the scheduling list as their deadlines approach, this technique should result in the preloading of most working sets, prior to the time the associated processes have to run. If a working set has to be swapped-out in order to make space for one that is to be swapped-in, either the unswapped process closest to the end of the scheduling list, or the process that blocked least recently, will be selected for swapping-out.[57] If the unswapped process that blocked least recently has been blocked for more than some threshold amount of time, it will be the one selected for swapping-out.

For the processes which are currently active, techniques are required for determining which pages belong to their working sets. At the time that a new process is created, there is no execution history available to indicate what its initial working set ought to be. In order to reduce the number of initial page faults, and hence improve the efficiency of process "loading" and startup, TDVM will allow an initial working set to be specified. This initial working set could either be determined heuristically (for example, *x* percent of the User Text and *y* percent of the User Data), or it could be based on information obtained through previous executions of the process.

As a process runs, TDVM must dynamically adjust the process' working set, to track the process through different phases of its execution. The primary way that pages get added to a working set is through page faults, i.e. as new pages are accessed they get paged-in "on demand". If TDVM detects sequential paging activity in one of the data segments of a process' address space, it can attempt to do sequential pre-paging. This technique can be very effective in improving the performance of sequentially accessed File Arobjects, and in many other situations as well. One other way that pages

[57] The scheduling list will contain all processes which are "sleeping" (due to *Delay* or *Alarm* primitives), but any processes blocked for other reasons (such as *Accept* primitives) will not appear in that list.

can get added to a working set is if the working set window size is increased.[58] When TDVM detects an increase in the window size (by checking the value in the scheduling list, obtained through the TDS *Schedule* primitive), it must scan through the "last accessed times" of all pages in the address space. This will allow it to determine which pages should be added to the working set (paged-in), in addition to those already there.

The removal of pages from a working set occurs when pages fall outside of the working set window. This can happen either as a result of the process' virtual time advancing more than *W* units since one or more of the working set pages were last accessed, or as a result of the working set window size decreasing. A decrease in the window size is detected in the same manner as an increase, but in this case only the current working set pages need be scanned to determine which, if any, are to be removed from the working set. Determining the last accessed times for pages of the working set can only be done approximately, assuming no specialized virtual memory hardware other than USED flags is available. It is accomplished by scanning the USED flags of the working set pages every time *w* units of virtual time have passed.[59] For every page in which the USED flag is set, that flag is cleared and the last accessed time is updated to the current virtual time. For any page in which the USED flag was clear, the last accessed time is checked to see if it falls outside of the working set window. If so, that page is removed from the working set.[60]

One other major responsibility of the TDVM Subsystem is management of the free page frame pool. A free page is any primary memory page which does not belong to one of the currently active working sets. Such a page can be in any one of three different states:

1. On the Page-Out List: It corresponds to an existing page in a secondary memory page set, but it has been modified and hence must be written back to the page set before it can be reused.

2. On the Reclaim List: It corresponds exactly to an existing page in a secondary memory page set, and hence it is free to be reused without first writing it back to the page set.

3. On the Free List: It is completely free to be reused, since it does not correspond to any existing page in a secondary memory page set.

---

[58] This happens when TDS determines that it has now become more important to avoid page faults, because the process is nearing its deadline and the penalty for missing the deadline is high.

[59] Recall that *W*, the working set window size, is some multiple of *w*, the base window size.

[60] Note that, in practice, working set scans could be initiated at the time of processor rescheduling decisions (calls to *Schedule*), assuming that such decisions occur at intervals of at most *w* units of real time. In that case, the current process' working set need only be scanned if the process has accumulated at least *w* units of virtual time since it was last scanned.

When a page fault occurs, TDVM checks the Page-Out and Reclaim Lists to see if the required page is already in primary memory, and hence can be quickly "reclaimed". If so, the time and expense of reading the page from its secondary memory page set can be avoided. Otherwise a free page must be selected to hold the contents of the page as it is read from the page set. Pages are selected first from the Free List, but if it is empty, then they will be selected from the Reclaim List. The Reclaim List is maintained in FIFO order, so that those pages which have been free for the longest time will be selected first for reuse. If the Reclaim List is also empty, then a page from the Page-Out List will have to be selected. Of course, that page will have to first be written back to its corresponding page set, before it can be reused. The Page-Out List, like the Reclaim List, is maintained in FIFO order, so the page that has been free the longest will be the first to be reused. In the event that no free page can be found, i.e. the Page-Out List is also empty, then TDVM must initiate the swapping-out of a process in order to make some free pages available.

To help avoid the situation of having to page-out or swap-out at page fault time (i.e. to help reduce the elapsed time for handling a page fault), TDVM attempts to maintain a mimimum number of pages in the Free List and Reclaim List combined. When the number of available pages in these lists drops below a threshold (approximately five percent of the primary memory page frames), TDVM will initiate page-out operations in order to move some pages from the Page-Out List to the Reclaim List. If there are still not enough free pages available, then TDVM will initiate a swap-out operation in order to free all of the working set pages from some process (the least urgent one). Note that checking the free page threshold and initiating page-out and swap-out operations can be done at the time of processor rescheduling decisions, just as for the case of initiating working set scan operations.

6.10.2 Time-Driven Virtual Memory Subsystem Primitives

The TDVM Subsystem provides the following set of primitives, primarily for use by the Arobject/Process Management (A/PM) Subsystem:

```
val = VMPrePage(asid, vpa, nvp)
val = VMLock(asid, vpa, nvp)
val = VMUnlock(asid, vpa, nvp)

pid = VMPageFault(asid, vpa)
pid = VMSchedule()

val = VMFlush(asid, vpa, nvp)
val = VMSwap()
gpsid = VMFreeze(asid)
val = VMUnfreeze(gpsid [, asid])

val = VMMove(asid, vpa, desired-vpa)
val = VMZero(asid, vpa, nvp)

val = VMDestroy(asid)
val = VMFree(asid, vpa)

val = VMStatus(statusbuffer [, asid])
val = VMRestart()
```

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

PID pid — The process ID of the process which is to be run at this time.

GPSID gpsid — Global Page Set ID, which identifies the *temporary* page set containing the frozen (swapped) address space descriptor information. It includes the ID of the logical disk containing the page set, the page set type (TEMPORARY), and the unique ID of this page set within the logical disk.

ASID asid — Address Space ID, which identifies the address space to be operated upon. The corresponding process ID can be easily obtained from an ASID, and vice versa.

VIRT-PAGE-ADDRESS vpa, desired-vpa
A virtual page address within an address space.

INT nvp — The number of virtual pages involved in the operation.

VMSTATUSBUFFER *statusbuffer
The buffer address for returning status information.

On Error: Error conditions are indicated by the use of special return values. The details concerning the precise nature of an error condition are provided in the Kernel Error Block.

The *VMPrePage* primitive initiates the paging-in of the specified set of virtual pages (*nvp* pages, beginning with page *vpa*), in address space *asid*. This is useful for setting up an initial working set of pages for a newly created process, since otherwise there is no execution history to indicate what the working set ought to be. The *VMLock* primitive is similar to *VMPrePage* in that it causes the specified set of virtual pages to be paged-in, if not already in primary memory. However, *VMLock* is synchronous in the sense that it will not return until all of the specified pages are actually present in primary memory. Furthermore, those pages will be "locked" in primary memory, i.e. they will not be considered eligible for removal from the address space's working set, regardless of how long it has been since they were last accessed. *VMUnlock* will "unlock" the specified set of virtual pages, making them eligible for removal from the working set, and allowing the corresponding physical page frames to be subsequently reused.

VMPageFault is the means by which the TDVM Subsystem is notified of page faults. It is assumed that the specified virtual page (*vpa*) is indeed one of the allocated pages of address space *asid*, and hence the page fault is truly due to accessing a valid page that isn't resident in primary memory (rather than a protection or invalid address fault). *VMPageFault* first checks to see if the missing page is already present somewhere in primary memory. This would be the case if the page had been in primary memory earlier, was "freed" due to inactivity, but the associated page frame had not yet been reused. It would also be the case if the page is shared *and is already resident* as part of another process' working set. If the missing page is found in primary memory, the page fault is handled very quickly, with no need to switch processes. In this case *VMPageFault* will return the ID (*pid*) of the current process, which is the one that encountered the page fault. Otherwise *VMPageFault* must initiate a page-in operation. Since reading a page from a secondary memory page set can take tens of milliseconds (or even hundreds of milliseconds if paging-out is required before space is available for the new page), a process switch is usually desireable when paging-in is required. In such cases, *VMPageFault* will return the ID of the process which should now run in place of the faulting process.

VMSchedule is intended to be the primary means by which the Arobject/Process Manager determines which process is to be running at the present time. *VMSchedule* calls the *Schedule* function of the Time-Driven Scheduler, to obtain the ordered list of processes which are to be run next. From this list the TDVM Subsystem can determine which address space working sets will be required in primary memory in the near future, allowing it to initiate any necessary page-in operations. Related to this, *VMSchedule* is also responsible for scanning the working set pages of the current process, to see if any of them have not been accessed recently and can thus be removed from the working set. This scanning operation is only initiated if the process has accumulated sufficient virtual (CPU) time since its working set was last scanned. *VMSchedule* returns the ID of the process which is

to run now. It guarantees that the address space for that process is the active one (see the description of *ASActivate* in Section 6.2.3.5).

The *VMFlush* primitive ensures that the specified set of virtual pages from the given address space are all "clean", i.e any primary memory images of these pages are paged out, if they have been flagged as MODIFIED. *VMFlush*, like *VMLock*, is synchronous. It will not return until all of the modified pages have been written out to their corresponding page sets. *VMFlush* is primarily intended to support the *FlushPermanent* primitive of the Arobject/Process Manager (see Section PRIVOBJSEC). It is also used when "deactivating" arobjects, to ensure that all modifications have been flushed out to the appropriate page sets before the address spaces are destroyed. Unless directed to flush modified pages through *VMFlush*, the TDVM Subsystem will only initiate page-out activity when primary memory begins to fill up, and some of the modified but not recently accessed page frames have to be reused.

VMSwap provides a mechanism through which the Arobject/Process Manager can explicitly request the TDVM Subsystem to "swap out" one of the existing but currently idle address spaces. Ordinarily, the TDVM Subsystem will only swap out an address space if the combined working sets of all of the existing address spaces overflow primary memory. In that case TDVM calls the Time-Driven Scheduler's *RequestSwapList* function, to obtain the ordered list of swappable processes. TDVM selects the most appropriate candidate from this list.[61] All of the pages from the selected process' working set are released, so that they can be paged-out and reused as required. TDVM then writes all of the relevant address space descriptor information (including the list of working set pages) into a *temporary* page set.[62] The address space is then destroyed (using *ASDestroy*), which frees the associated address space resources for use by others. This freeing of address space resources is the primary reason for making *VMSwap* available to the Arobject/Process Manager. It provides a mechanism for recovering from "space exhausted" problems when creating or growing address spaces.

VMFreeze is quite similar to *VMSwap*, except that it allows a specific address space (*asid*) to be selected for "swapping". If that address space is already swapped out, the global page set ID (*gpsid*)

---

[61] The least "urgent" process, for which the expected continuation time is farthest in the future and which hasn't already been swapped out, is selected. However, processes containing locked address space pages are not considered eligible for swapping.

[62] It is not necessary to record all of the address space information. Only the Private Region segment descriptors and the EXISTS/COPIED flag for each page in those segments need be saved. If this was the last process from the associated arobject which was still resident on this node and not yet swapped out, then the Shared Region segment descriptors and the corresponding EXISTS flags will also have to be saved.

of the *temporary* page set containing the address space descriptor is returned, after first ensuring that all MODIFIED pages from that address space have been flushed.[63] If the specified address space is not currently swapped out, it is flushed and swapped (even if it contains locked pages), and the *gpsid* of the *temporary* page set containing the address space descriptor is returned. *VMFreeze* provides support for the Arobject/Process Management Subsystem's *FreezeArobject* and *FreezeProcess* primitives (see Section APMSEC). It is also used when migrating processes from one node to another.

Following a *VMFreeze* operation, the TDVM Subsystem removes all record of address space *asid*, except for the page sets on secondary memory. The *VMUnfreeze* primitive can be used to restore a previously frozen address space (indicated by *gpsid*, the ID of the page set containing the address space descriptor), to the swapped state (see Figure 6-33). *VMUnfreeze* provides support for the *UnfreezeArobject* and *UnfreezeProcess* primitives of the A/PM Subsystem (see Section APMSEC). Optionally, an address space can be given a new ID (*asid*) at the time it is unfrozen. This is useful when migrating processes, since it allows the address space to be associated with a new process ID on the new node.

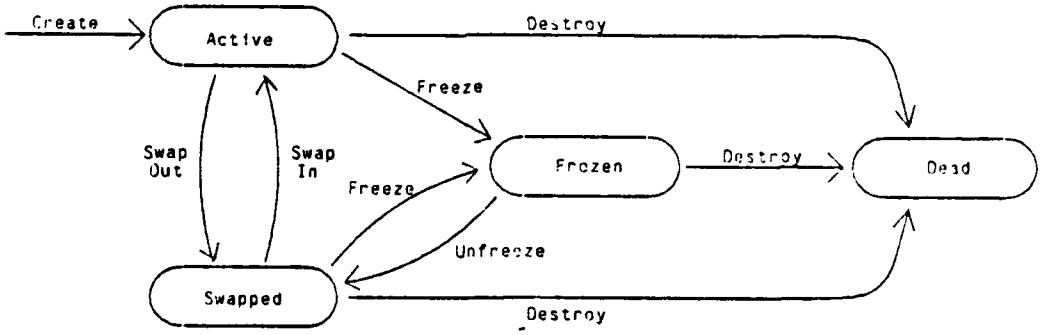


Figure 6-33: Life Cycle of an Address Space

VMMove is the Arobject/Process Manager's interface to the *ASMove* function, which changes the location of an existing segment in address space *asid* from *vpa* to *desired-vpa*. This function must go through the TDVM Subsystem, so that TDVM can update any references it may have to the affected virtual pages. See Section 6.2.3.5 for more details concerning the *ASMove* primitive. *VMZero*

[63] When an address space is swapped, all of its MODIFIED pages are flagged for paging-out but are not flushed immediately. A frozen address space, on the other hand, is guaranteed to have been completely flushed.

provides an efficient mechanism for "zeroing" complete pages of an address space. It does this by clearing the EXISTS flags for the specified virtual pages, so that the next time any of these pages are accessed they will first be zero-filled. If the pages to be zeroed already exist on the corresponding page set (the EXISTS flags were set), then *VMZero* will also call the appropriate *PSZero* or *APSZero* primitive to zero (or free) the pages from the page set. One of the main purposes of the *VMZero* primitive is to provide efficient support for the *ZeroFile* primitive, which is provided by the File System Client Interface (see Section 6.6).

VMDestroy and *VMFree*, like *VMMove*, are the A/PM Subsystem's interface to the corresponding Address Space Management Primitives (*ASDestroy* and *ASFree*). *VMDestroy* completely destroys the specified address space (*asid*), while *VMFree* deletes a segment (indicated by *vpa*) within address space *asid*. These functions must go through the TDVM Subsystem, so that TDVM can remove any references it may have to the affected address space or segment, and free any corresponding primary memory page frames. See Section 6.2.3.5 for more details concerning the *ASDestroy* and *ASFree* primitives.

The *VMStatus* primitive is used to obtain general information concerning the current status of the TDVM Subsystem. This includes the number of active and swapped address spaces, the average size of the address space working sets, the number of free pages available, and the total number of page-in, page-out, and page-reclaim (avoided or fast page-in) operations. This information can be used for monitoring the system's virtual memory activity, so that the TDVM Subsystem can be tuned to provide better performance. If an optional address space ID (*asid*) is specified, then *VMStatus* will return virtual memory information related to that particular address space. This includes its status (SWAPPED or ACTIVE), working set size, and page-in, page-out, and page-reclaim statistics.

The *VMRestart* primitive is the initialization operation for the TDVM Subsystem (on a particular node). It is assumed to be called automatically as the first TDVM operation, upon restarting (rebooting) a failed node. Its sole purpose is to create the TDVM worker (kernel) processes, and initialize the virtual memory management data structures. These TDVM internal processes and data structures are described in the following section.

6.10.3 Components of the Time-Driven Virtual Memory Subsystem

The TDVM Subsystem is implemented as a kernel arobject, with a separate instance on each node of the distributed computer system. Each instance is solely responsible for the management of the primary memory page frames on its own node, and has no need to communicate or cooperate with any of its peers on other nodes. Each instance of the TDVM Subsystem consists of three processes

and five main data structures, as illustrated in Figure 6-34. The TDVM Manager process provides almost all of the subsystem's functionality, and is the only process which can access and modify the main data structures. The sole purpose of the Page-In Worker and Page-Out Worker processes is to allow paging (page set read and write) operations to be performed asynchronously with other TDVM Subsystem processing. The two worker processes are basically surrogates for the TDVM Manager. They wait for page set operations to be completed on behalf of the TDVM Manager, allowing the Manager to continue handling other virtual memory related operations.

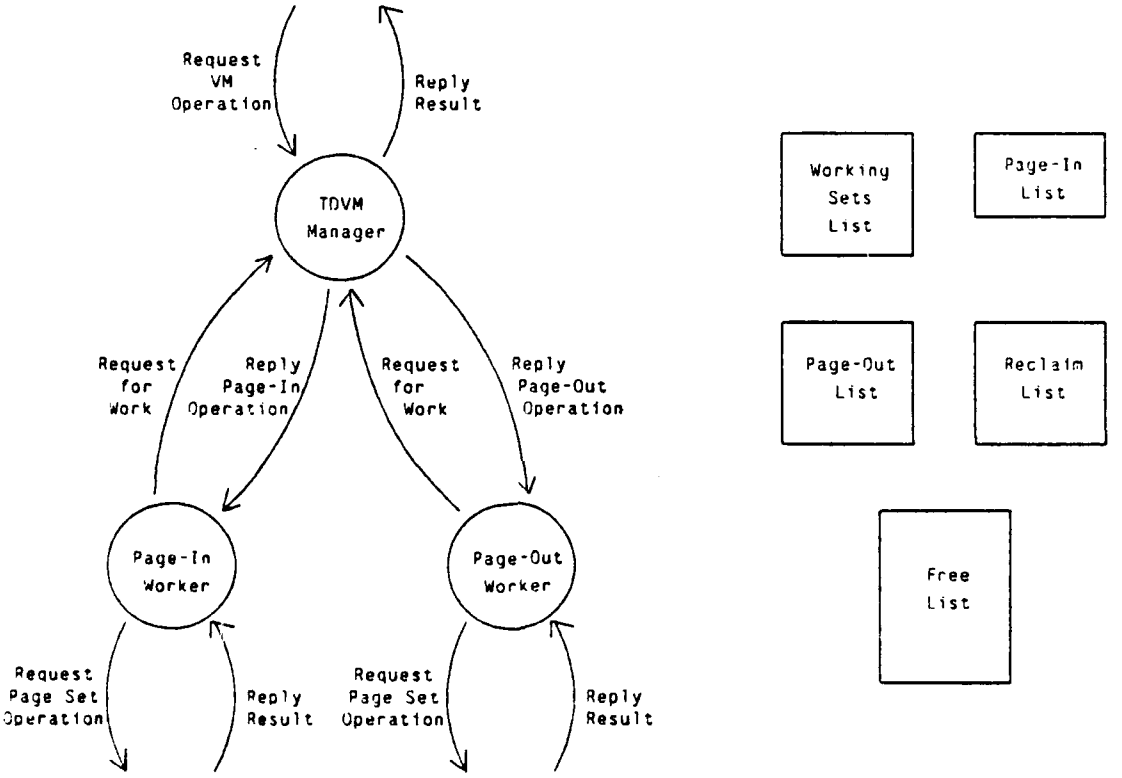


Figure 6-34: Components of the Time Driven Virtual Memory Subsystem

The first main data structure indicated in Figure 6-34 is the Address Space Working Sets List. This data structure is illustrated in more detail in Figure 6-35. The Address Space List contains an entry for every address space that is known to the TDVM Subsystem (whether it is currently active or swapped out). Each entry indicates the current *state* of the address space (ACTIVE or SWAPPED). If SWAPPED, the global page set ID (*gpsid*) of the *temporary* page set containing the address space's

descriptor is provided. See the second entry in the Address Space List in Figure 6-35 for an example. Otherwise the entry contains information concerning the address space's current working set of virtual pages.

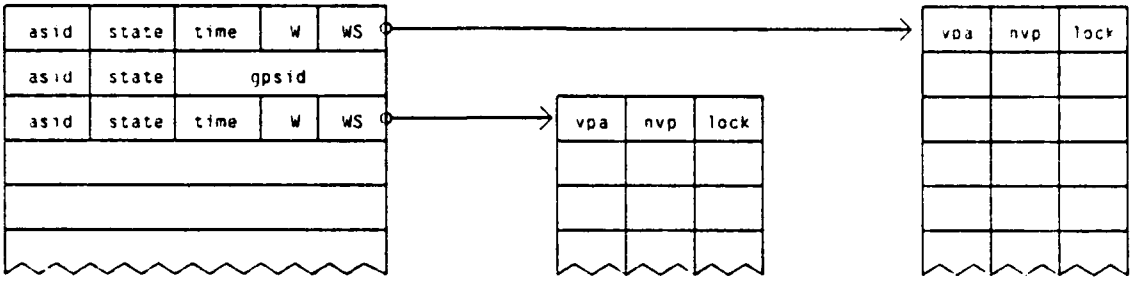


Figure 6-35: Address Space Working Sets List

For ACTIVE address spaces, the "last scan time" (*time*) and working set window size (*W*) are provided, in addition to the actual list of virtual pages which comprise the working set. The last scan time is the virtual (CPU) time at which the address space was last scanned to determine which pages belong to the working set (were accessed recently). The *W* parameter defines "recently", by specifying the virtual time frame during which a page must have been accessed, in order to be considered a member of the current working set. The list of working set pages is arranged in clusters, where each cluster is indicated by a starting address (*vpa*) and its size (*nvp*). Due to the "locality of references", clustering of working set pages is expected to be quite common. Thus, this representation of working sets will save space, and it will also improve the efficiency of page-in, page-out, swap-in, and swap-out operations, since entire clusters can be read and written at a time. Also associated with each cluster is a *lock* flag, which indicates whether or not those pages are locked in primary memory (as a result of a previous *VMLock* primitive).

The second main data structure of the TDVM Subsystem is the Page-In List. This list is actually maintained as two separate lists, one for urgent page-in operations (those resulting from page faults), and another for the less urgent "pre-paging" operations (including the swapping in and expanding of the working sets of address spaces which have increased in "urgency"). The use of two lists allows the urgent page-in operations to be given precedence over pre-page operations. The two Page-In Lists have identical structures, as illustrated in Figure 6-36.

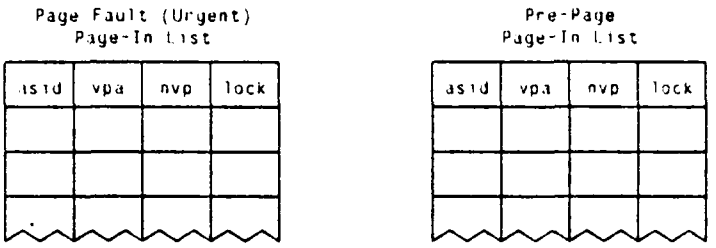


Figure 6-36: Page-In Lists

Each entry in a Page-In List indicates the address space (*asid*) and the virtual page (*vpa*) of a virtual page that is to be paged-in. The corresponding page set and page number can then be determined using the *ASGetGPSID* or *ASGetSTE* primitives (see Section 6.2.3.5). Each entry can also indicate a range or cluster of *nvp* virtual pages, beginning with virtual page *vpa*. Normally, *nvp = 1* for entries in the Urgent List, since this will allow the corresponding process to continue executing as quickly as possible.[66] However, the clustering of page-in operations in the Pre-Page List will help improve the overall paging throughput. Each entry in a Page-In List also contains a *lock* flag, which indicates whether that cluster of pages is to be locked into the corresponding address space's working set, after it has been paged-in.

Two of the other main data structures of the TDVM Subsystem are the Page-Out List and the Reclaim List. These two structures are closely related in that they both contain information about primary memory page frames which no longer belong to any address space's working set. The only difference between the two lists is that the pages in the Page-Out List have been modified, and hence they must be written back to their corresponding page sets before they can be reused. The Page-Out List and the Reclaim List each have the same logical structure, as illustrated in Figure 6-37.

Each entry in the Page-Out List or Reclaim List represents a cluster of *npages* physical page frames, beginning with *ppa*. This cluster corresponds to the *npages* of page set *gpsid*, beginning with page *pnum*.[67] The Page-Out List and Reclaim List together allow the TDVM Subsystem to quickly "reclaim" pages which have not yet been reused, thus avoiding unnecessary page-in operations. Although

[66] Some experimentation is needed to determine in which circumstances, if any, it might be desireable to perform clustered urgent page-in operations.

[67] Once again the clustering of pages, especially in the Page-Out List, should help improve paging throughput.

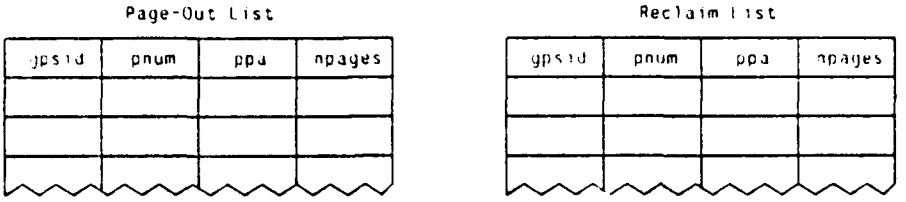


Figure 6-37: Page-Out and Reclaim Lists

conceptually they are simple lists (as illustrated in Figure 6-37), in practice the Page-Out List and Reclaim List would be sorted into binary search trees, using (*gpsid*, *pnum*) as the key. This would make the searching for reclaimable pages, as well as the insertion of new pages into existing clusters very fast. In addition, each entry in the Page-Out List and Reclaim List would be threaded in FIFO order on its respective "reuse queue". This ordering would allow the least recently used pages to be selected for reuse, whenever the pool of "free" page frames was exhausted.

The final main data structure of the TDVM Subsystem is the Free Page Frame List, which is illustrated in Figure 6-38. Each entry in the Free List has a very simple structure, and represents a cluster of *npages* physical page frames, beginning with *ppa*. Each cluster is completely free to reuse, since it does not correspond to any existing virtual address space pages. In practice, the Free List would be sorted into a binary search tree, using *ppa* as the key. This would make the insertion of newly freed page frames into existing clusters very fast. In addition, each entry would be threaded on an appropriate list according to its cluster size (*npages*). There are 32 separate cluster size lists, one for each power of two. This arrangement helps reduce primary memory "fragmentation", allowing page-in clusters of the appropriate size to be found very quickly.

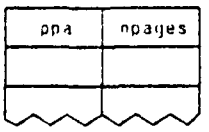


Figure 6-38: Free Page Frame List

6.10.4 Interaction With Arobject/Process Manager and Time-Driven Scheduler

The TDVM Subsystem interacts extensively with both the Arobject/Process Manager and the Time Driven Scheduler, on the local node. Figure 6-39 illustrates the primary "uses" relationships among these subsystems. A/PM interacts with TDVM in two main ways, either directly through the invocation of TDVM primitives, or indirectly through the manipulation of address spaces. Since most address space manipulations are actually performed by TDVM, and TDVM must maintain consistency between its own data structures and those of the Address Space Management Routines, A/PM is quite restricted in terms of the address space operations it is allowed to perform. In particular, certain operations (*ASDestroy*, *ASFree*, *ASMove*) must be invoked by A/PM via the corresponding TDVM primitives, as noted earlier. However, address space creation (*ASCreate*) and growth (*ASAllocate*, *ASExpand*) can both be handled directly by A/PM, since TDVM will learn about them automatically as the new address space and pages are used.

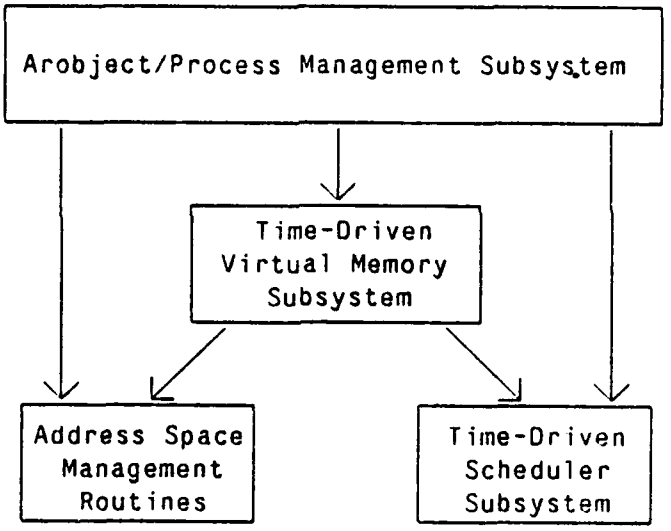


Figure 6-39: TDVM, A/PM, and TDS Subsystem Interactions

Besides address space manipulations, the other major area of A/PM and TDVM interaction is process scheduling. The determination of which process is the best choice to be running on a node at any given time is ultimately the responsibility of the Time-Driven Scheduler. However, TDS is not aware of the status of the process address spaces (ACTIVE or SWAPPED), and thus it could select a

process which TDVM has not yet had time to swap back in.[68] In order to coordinate properly between TDVM and TDS on the choice of which process to run at the present time, A/PM must always use the *VMSchedule* primitive, rather than directly invoke the TDS *Schedule* primitive. Figure 6-40 illustrates the normal sequence of events which occur when A/PM must reschedule the processor, due to the current process blocking or completing.

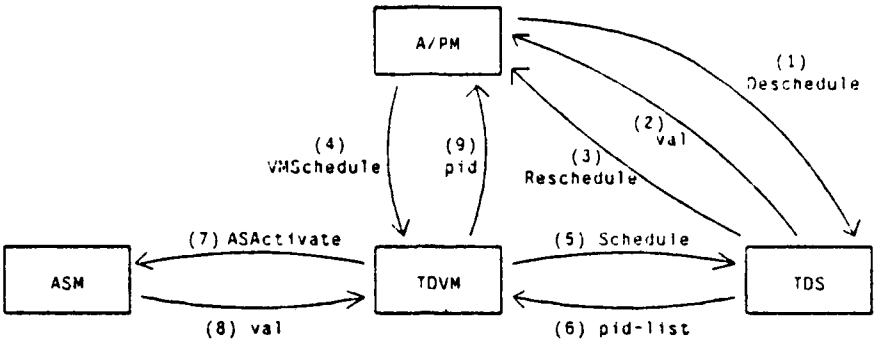


Figure 6-40: Normal Processor Rescheduling Sequence

A/PM first invokes the TDS *Deschedule* primitive, and after it completes (returns *val*) A/PM enters the "idle state", waiting for a "*Reschedule*" signal to indicate that another process is ready to run. When TDS has determined which process to run next, it sends the *Reschedule* signal (interrupt) to A/PM.[69] Upon receipt of the *Reschedule* signal, AP/M calls *VMSchedule* to determine exactly which process is to be run next, and to switch to its associated address space. *VMSchedule* itself calls the TDS *Schedule* primitive, to see which process the scheduler has selected. Assuming the selected process is not swapped out, TDVM activates the process' address space (by calling *ASActivate*), and returns the ID of the process (*pid*) to A/PM. A/PM then completes the switch to process *pid*, and sets it running.

If the process selected by the TDS *Schedule* primitive (the first process returned in *pid-list*) is currently swapped out, *VMSchedule* will initiate the swapping-in of that process, and select a different

[68] This should seldom happen, but it can occur if all of the most urgent processes block in rapid succession, waiting for various events. The best choice among the remaining "ready" processes could then be one that is still swapped out, since TDVM had not anticipated the need to run it so soon. Another way this can happen is if a process which was blocked for a long time and got swapped out suddenly becomes ready to run again, and it is immediately selected as the most urgent process.

[69] If another process is already available and ready to run, there should be essentially zero delay before TDS sends the *Reschedule* signal.

pid from *pid-list*. To be selected, a process must be the first one in *pid-list* which is not currently swapped out, awaiting completion of a page-in operation, or put on "*hold*" by TDS.[70] If no appropriate process can be found, *VMSchedule* returns BAD-PID to A/PM. This informs A/PM that it should return to the idle state, rather than switch processes at this time.

In addition to rescheduling the processor when the current process blocks or completes, preemptive rescheduling can also occur. Preemptive rescheduling due to an urgent process unblocking works almost identically to the normal rescheduling sequence outlined above, and illustrated in Figure 6-40. The only difference is that A/PM invokes the TDS *SetScheduleInfo* primitive (rather than *Deschedule*), upon the occurrence of an event for which a process is waiting (blocked). This informs TDS that the previously blocked process is now schedulable, so that TDS can determine when it should be run. In the meantime, A/PM can allow the process which was running at the time of the event to continue executing. If TDS determines that the newly unblocked process should be run immediately, it will send a *Reschedule* signal to A/PM. A/PM will then save the state of the current process, and initiate the standard *VMSchedule* sequence discussed above. Similarly, if at any point TDS wishes to preemptively reschedule the processor (because an urgent process has just completed its delay, or there has been some other significant change in the value functions of processes), it can do so by simply sending a *Reschedule* signal to A/PM.

Two other reasons for rescheduling the processor, of particular relevance to TDVM, are the occurrence of page faults and the completion of page-in and swap-in operations. The sequence of events which occurs upon page fault is illustrated in Figure 6-41. A process execution fault first comes to the attention of A/PM. If it is a page fault, A/PM saves the state of the current process, and invokes the *VMPageFault* primitive of TDVM. If a fast page reclaim is possible, TDVM handles it immediately and returns the *pid* of the current process, so that A/PM can continue its execution. Otherwise, TDVM initiates a page-in operation, and calls *Schedule* to determine which process should be run while waiting for the page-in to complete. Note that since the faulting process has not been *Descheduled*, it will still appear in the *pid-list* returned by *Schedule*, but it will be ineligible for selection because it is awaiting completion of the page-in operation. If no appropriate process is available, BAD-PID is returned to A/PM, indicating that the processor should remain idle at this time. Otherwise, the address space of the selected process is activated, using *ASActivate*, and its *pid* is returned to A/PM in order to set it running.

---

[70] The *hold* flags are returned along with the *pids* in *pid-list*. They indicate which processes are not to be run at present, either because they are still "sleeping", or because there is a penalty for running them too soon.

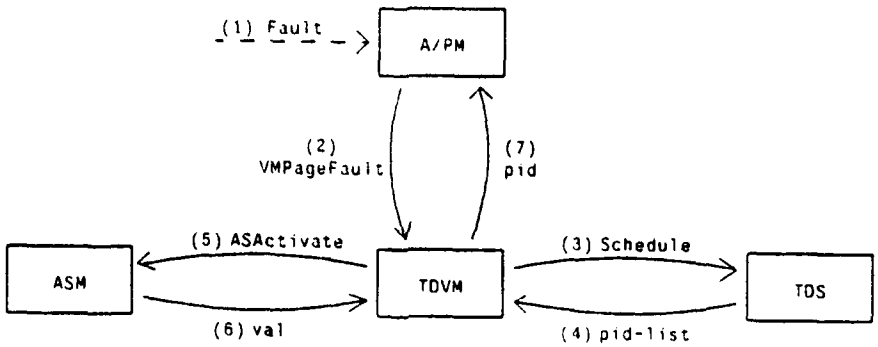


Figure 6-41: Page Fault Sequence

The sequence of events which occurs upon the completion of a page-in or swap-in operation is illustrated in Figure 6-42. TDVM is notified of such an event through a new "Request for Work" from its Page-In Worker process (see Figure 6-34). In response to this, TDVM initiates the standard (preemptive) rescheduling sequence (as discussed above), by sending a *Reschedule* signal to A/PM. Note that the process for which the page-in or swap-in operation has just been completed should still appear in the *pid-list* returned by *Schedule*, but now it will be eligible for selection as the process to be run next.

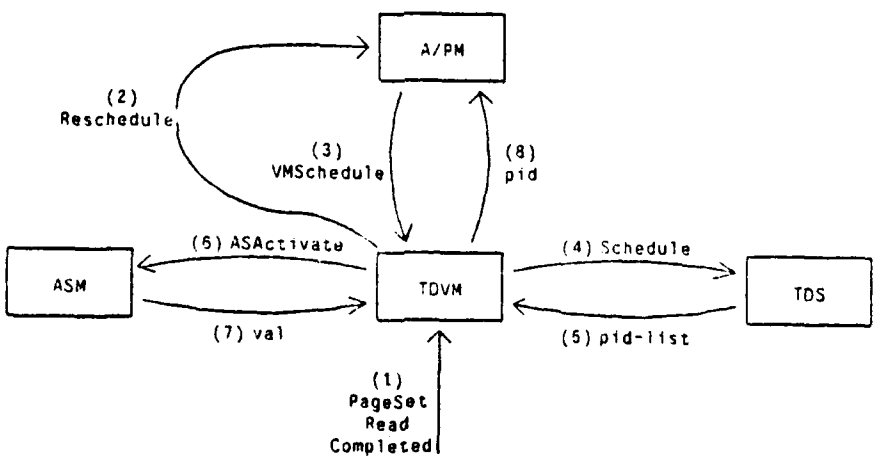


Figure 6-42: Page-In or Swap-In Completion Sequence

6.11 System Monitoring and Debugging Subsystem

The system monitoring and debugging subsystem provides various abilities to monitor and control behavior of cooperating arobjects and processes during the execution time. The monitoring and debugging manager exists on each node and has a special privilege to *freeze* or *unfreeze* an activity of the arobject or process.

6.11.1 Monitoring and Debugging Management

The actual operations for the monitoring and debugging operations are performed in the arobject/process or communication subsystems. For instance, *Freeze*, *UnFreeze*, *Fetch*, *Store* operations on an arobject/process is performed at the arobject/process manager. However, *Freeze* and *UnFreeze* operations for a specific node or for all applications are initiated at the monitoring and debugging manager. Similarly, monitoring on the communication activity is supported by the communication subsystem.

The following ArchOS primitives are supported for the monitoring and debugging management for a client.

---

val = **FreezeAllApplications()**
val = **UnfreezeAllAplicatons()**
val = **FreezeNode**(node-id)
val = **UnfreezeNode**(node-id)

BOOLEAN val — TRUE if the primitive was executed properly; otherwise FALSE.

NODE-ID node-id — The node id indicates the actual node which will be stopped.

A *FreezeAllApplications* primitive stops the entire activities of caller's application, and a *FreezeNode* primitive halts all of the client's activities in a specific node. To resume client's application, *UnfreezeAllApplications* or *UnfreezeNode* will be used.

6.11.2 Monitoring/Debugging Protocol

When a system-wide Monitoring/Debugging request such as *FreezeAllApplications* is issued, a Monitoring/Debugging Manager's worker becomes a coordinator among the other Monitoring/Debugging Managers and propagates the request to the others.

References

[Ada 83] *Ada Programming Language*
1983.

[Almes 83] Almes, G. T.; Black, A. P.; Lazowska, E. D.; Noe, J. D.
The Eden System: A Technical Review.
Technical Report 83-10-05, University of Washington, October, 1983.

[Bernstein 83] P. A. Bernstein, Goodman, N.; Hadzilacos, V.
Recovery Algorithms for Database Systems.
Technical Report TR-10-83, Harvard University, 1983.

[Bobrow 81] Bobrow, D. G.; Stefik, M.
The LOOPS Manual.
Technical Report KB-VLSI-81-13, Xerox Palo Alto Research Center, August, 1981.

[Comer 79] Comer, D.
The Ubiquitous B-tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Conway 67] Conway, Maxwell, and Miller.
Theory of Scheduling.
Addison-Wesley, 1967.

[Denning 68] Denning, P.J.
The Working Set Model for Program Behavior.
Communications of the ACM 11(5):323-333, May, 1968.

[Denning 80] Denning, P.J.
Working Sets Past and Present.
IEEE Transactions on Software Engineering SE-6(1):64-84, January, 1980.

[Garey 79] Garey, M. R.; Johnson, D. S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W. H. Freeman, San Francisco, 1979.

[Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
Operating Systems Review 13(5), December, 1979.

[Goldburg 83] Goldburg, A.; Robson, D.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley, 1983.

[Gray 77] James Gray.
Notes on Data Base Operating Systems.
Technical Report, IBM Research Laboratory, San Jose, 1977.

[Herlihy 84] Herlihy, M. P.
Replication Methods for Abstract Data Types.
PhD thesis, MIT Laboratory for Computer Science, 1984.

[Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System -- An Overview of its Objectives and Approach.
IEEE Distributed Processing Technical Committee Newsletter, Special Issue on Distributed Operating Systems , June, 1984.

[Jensen 85] E. Douglas Jensen, H. Tokuda, et al.
Functional Description of ArchOS (Archons Operating System).
January, 1985
Technical Report for RADC, Contract No. F30602-84-C-0063, Department of Computer Science, Carnegie-Mellon University.

[Kernighan 78] Brian W. Kernighan, Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[Lazowska 31] Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M. J., Fowler, R. J., and Vestal, S. C.
The Architecture of the Eden System.
Operating Systems Review 15(5):148-159, December, 1981.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Locke 85] C. Douglass Locke, Tokuda, H; E. Douglas Jensen.
A Time-driven Scheduling Modle for Real-Time Operating Systems.
To appear in Proceedings of Real-time Systems Symposium , 1985.

[Mckendry 85] M. S. Mckendry; Herlihy, M.
Time-driven Orphan Elimination.
Technical Report CMU-CS Tech Report CMU-CS-85-138, Department of Computer Science, Carnegie-Mellon University, July, 1985.

[Mitchell 82] Mitchell, J., and Dion, J.
A Comparison of Two Network-Based File Servers.
Communications of the ACM 25(4):233-245, April, 1982.

[Moss 81] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April, 1981.

[Sha 83] Sha, L., Jensen E. D, Rashid, R. F., and Northcutt, J. D.
Distributed Co-operating Processes and Transactions.
Proceedings of ACM SIGCOMM symposium , 1983.

[Sha 84] Sha, Lui.
Synchronization in Distributed Operating Systems.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1984.
in preparation.

[Sha 85] Sha, L.

PhD thesis, Carnegie Mellon University, 1985.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
Journal of the Association of Computing Machinery 27(1), January, 1980.

[Sturgis 80] Sturgis, H., Mitchell, J., and Israel, J.
Issues in the Design and Use of a Distributed File System.
Operating Systems Review 14(3):55-69, July, 1980.

[Tokuda 83a] Tokuda, H; Manning, E. G.
An Interprocess Communication Model for a Distributed Software Testbed.
Proceedings of ACM SIGCOMM symposium , 1983.

[Tokuda 83b] Tokuda, H , Radia, S. R., and Manning, E. G.
Shoshin OS, a Message based Distributed Operating System for a Distributed Software Testbed.
Proceedings of the 16th Hawaii Int. Conf. on System Sciences , 1983.

[Tokuda 85] Tokuda, H.
A Compensatable Atomic Objects in Object oriented Operating Systems.
To appear in Proceedings of Pacific Computer Communicaton Symposium , 1985.

[Weinreb 81] Weinreb, D.; Moon, D.
Lisp Machine Manual.
Technical Report, MIT Artificial Intelligence Laboratory, 1981.

[Wulf 81] Wulf, W. A., Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System
McGraw Hill, 1981.

END
DATE
FILMED

8-12-87